The impact of treewidth on ASP grounding and solving of Answer Set Programs

Bliem, B.

2020

Bliem , B , Morak , M , Moldovan , M & Woltran , S 2020 , ' The impact of treewidth on ASP grounding and solving of Answer Set Programs ' , Journal of Artificial Intelligence Research , vol. 67 , pp. 35-80 . https://doi.org/10.1613/jair.1.11515

http://hdl.handle.net/10138/310289
https://doi.org/10.1613/jair.1.11515

publishedVersion

Journal of Artificial Intelligence Research 67 (2020) 35-80 Submitted 05/2019; published 01/2020

The Impact of Treewidth on Grounding and Solving of Answer Set Programs

Bernhard Bliem BERNHARD.BLIEM@GMAIL.COM
University of Helsinki
Constraint Reasoning and Optimization Group
Helsinki, Finland

Michael Morak MICHAEL.MORAK@AAU.AT
University of Klagenfurt
Semantic Systems Group
Klagenfurt, Austria

Marius Moldovan MOLDOVAN@DBAI.TUWIEN.AC.AT
Stefan Woltran WOLTRAN@DBAI.TUWIEN.AC.AT
TU Wien
Databases and Artificial Intelligence Group
Vienna, Austria

Abstract

In this paper, we aim to study how the performance of modern answer set programming (ASP) solvers is influenced by the treewidth of the input program and to investigate the consequences of this relationship. We first perform an experimental evaluation that shows that the solving performance is heavily influenced by treewidth, given ground input programs that are otherwise uniform, both in size and construction. This observation leads to an important question for ASP, namely, how to design encodings such that the treewidth of the resulting ground program remains small. To this end, we study two classes of disjunctive programs, namely guarded and connection-guarded programs. In order to investigate these classes, we formalize the grounding process using MSO transductions. Our main results show that both classes guarantee that the treewidth of the program after grounding only depends on the treewidth (and the maximum degree, in case of connection-guarded programs) of the input instance. In terms of parameterized complexity, our findings yield corresponding FPT results for answer-set existence for bounded treewidth (and also degree, for connection-guarded programs) of the input instance. We further show that bounding treewidth alone leads to NP-hardness in the data complexity for connection-guarded programs, which indicates that the two classes are fundamentally different. Finally, we show that for both classes, the data complexity remains as hard as in the general case of ASP.

1. Introduction

Answer set programming (ASP) (Marek & Truszczyński, 1999; Brewka, Eiter, & Truszczynski, 2011; Gebser, Kaminski, Kaufmann, & Schaub, 2012) is a well-established logic programming paradigm based on the stable model semantics. Its main benefit is an intuitive, declarative language, and the fact that, generally, each answer set of a given logic program describes a valid solution of the original problem. Solving ASP programs is often defined as a two-step process. First, a (usually fixed) encoding for a given problem is written in the language of non-ground ASP. This encoding, together with a set of input facts repre-

senting the actual problem instance, gets passed to a *grounder* which transforms it into an equivalent propositional ASP program. In the second step, this ground program is then evaluated by a *solver*. Such ASP solvers are now readily available (Gebser, Kaufmann, & Schaub, 2012; Alviano, Dodaro, Leone, & Ricca, 2015; Elkabani, Pontelli, & Son, 2005; Leone, Pfeifer, Faber, Eiter, Gottlob, Perri, & Scarcello, 2006, inter alia) and have made huge strides in efficiency.

This leads to the following interesting practical question: What is the relationship of solver efficiency and different parameters of the ground input program, and how is the solving time influenced by these parameters? On the theoretical side, computational complexity investigations were carried out for the classical parameter of input size (Eiter & Gottlob, 1995; Truszczyński, 2011; Dantsin, Eiter, Gottlob, & Voronkov, 2001), while several structural parameters were studied in the field of parameterized complexity (Gottlob, Pichler, & Wei, 2010; Pichler, Rümmele, Szeider, & Woltran, 2014; Fichte & Szeider, 2015; Fichte, Kronegger, & Woltran, 2017). This has also led to specialized implementations that try to explicitly exploit these parameters (Jakl, Pichler, & Woltran, 2009; Fichte, Hecher, Morak, & Woltran, 2017). While these theoretical investigations provide us with valuable insight into the problem of ASP solving, it is not obvious what conclusions can be drawn for the actual practical solving performance of today's top-of-the-line ASP solvers. It would be interesting to see how current CDCL-based solvers are influenced, in practice, by variations in such structural parameters and whether guidelines for ASP modeling can be derived from such interactions. One of the few results in this direction is the discovery of a strong correlation between the rule-to-atom ratio of a ground ASP program and the solving time (Zhao & Lin, 2003), a property that carries over from similar studies for SAT (Selman, Mitchell, & Levesque, 1996). Some more recent studies on phase transitions in ASP also deals with this topic (Wen, Wang, Shen, & Lin, 2016; Amendola, Ricca, & Truszczynski, 2017). Beside these results, however, the practical impact of structural parameters on solving time has not, in the authors' opinion, received adequate attention in the literature.

1.1 Contributions

In this paper we focus on the parameter of treewidth, a measure of how closely a ground ASP program structurally resembles a tree. Our goal is to study how the performance of modern ASP solvers is influenced by the treewidth of the given ground input program and to investigate the consequences of this relationship. To this end, our first main contribution is to carry out an extensive experimental evaluation of two top-of-the-line ASP solver implementations and investigate how the solving performance behaves when the solvers are presented with hard instances of uniform size and construction, but variable treewidth, using a carefully crafted ASP problem encoding and adequately generated, tree-like instances. Our experiments show that the solving time for programs of the same size and construction indeed increases drastically with the treewidth. This is an interesting result, which shows that similar results for SAT solvers and resolution-width (Atserias, Fichte, & Thurley, 2011) do indeed carry over to the more complex world of ASP. Our observations suggest that, when encoding problems in (non-ground) ASP, it is not only important that the resulting ground program is small, but also that its treewidth is kept as small as possible, given a set of input facts. Let us illustrate this on a small example.

Example 1. Reachability can be modeled in different ways using ASP. One way would be to model the transitive closure of a graph as follows (where `e` is the predicate representing graph edges and `r` the predicate to mark reachable vertices):

```
t(X,Y) :- e(X,Y).
t(X,Z) :- t(X,Y), e(Y,Z).
r(Y) :- t(X,Y), start(X).
```

However, when used as a sub-program that the grounder has to instantiate, such an encoding causes any two (connected) vertices in the input graph to appear together in a rule after grounding (in place of the variables `X` and `Z` in the second rule). This then causes the graph representation of the ground program to contain a clique whose size equals the number of vertices of the original input graph, resulting in a high treewidth. Conventional wisdom in ASP would recommend the following encoding:

```
r(X) :- start(X).
r(Y) :- e(X,Y), r(X).
```

Here, not only is the grounding smaller, but also the treewidth decreases dramatically. In fact, it now solely depends on the treewidth (and not the size) of the input graph. △

This example illustrates that the way a problem is encoded can influence the treewidth of the ground program considerably. Due to the grounding step, however, it is not obvious at the time of writing a non-ground ASP encoding how to achieve a low-treewidth grounding and the benefits that come with it. This is as opposed to, for example, SAT formulas that can be generated directly while keeping treewidth in mind.

The second main contribution of this paper addresses this issue and allows us to leverage the treewidth-sensitivity of ASP solvers. We define classes of disjunctive logic programs that aim to capture the intuition described above; that is, to guarantee that treewidth of the grounding of a given program stays small, as long as the input instance has small treewidth as well. In order to achieve this goal, we propose two classes. Firstly, the class of *guarded* programs is inspired by similar classes defined in the datalog world; see e.g. Gottlob, Grädel, and Veith (2002). The main idea is that each rule body must contain a so-called *guard* atom, that is, an extensional atom in the positive body of the rule that contains every variable that appears in the rule. Our second class, namely the class of *connection-guarded* programs, relaxes the condition of guarded rules in the sense that the guard can be divided into several atoms under the condition that these guard atoms are sufficiently connected via their variables. Our main results show that both classes guarantee that the treewidth of a program after grounding does not increase arbitrarily but only depends (a) on the treewidth of the input facts, in the case of guarded programs; (b) on the treewidth and degree of the input facts, in the case of connection-guarded programs. In order to prove these results, we make use of the notion of MSO transductions (Courcelle & Engelfriet, 2012), a formal framework for graph-to-graph transformations. We use this framework to formally represent the grounding process and investigate its influence on the treewidth. To our knowledge, this is the first time that this technique has been used in the context of ASP.

As a third contribution, we provide an analysis of our classes in terms of computational complexity. In particular, we are interested in the data complexity, that is, the complexity of checking whether, given a fixed non-ground program and an input structure represented as a set of facts, the combined program has at least one answer set. While our results on

grounding imply FPT results for inputs parameterized by treewidth (and degree, in case of connection-guarded programs), we show that programs in our classes are, at the same time, expressive enough to encode relevant problems from the second level of the polynomial hierarchy in general. In fact, we show that data complexity for both classes remains Σ_2^{P}-complete, even for inputs parameterized by degree only. Finally, we show that our two proposed classes are indeed different by proving that the data complexity of answer set existence is in FPT for guarded programs, but NP-hard for connection-guarded programs if the input is parameterized by treewidth.

1.2 Structure

The remainder of the paper is structured as follows. In Section 2, we give relevant definitions for ASP, treewidth, and MSO transductions. Section 3 deals with our first main contribution, the experimental evaluation of solver performance with respect to treewidth, and shows that there is a significant correlation. Section 4 presents our second main contribution, namely, proposing the classes of guarded and connection-guarded ASP programs that aim to preserve the treewidth of the given instance after grounding. Section 5 contains the complexity analysis and Section 6 discusses related work and implications of our results for ASP solving and modeling. Finally, we conclude the paper with some final remarks in Section 7.

2. Preliminaries

This section provides an overview of the definitions and basic notions and constructs used throughout the rest of the paper.

2.1 Graphs, Tree Decompositions, and Treewidth

We assume all graphs to be undirected, simple, and ordered, which means that there is an arbitrary but fixed total order over the vertices of the graph. For a graph G, $V(G)$ denotes the set of vertices and $E(G)$ the set of edges. A graph H is a *minor* of a graph G if H can be obtained from G by deleting edges or vertices, or by contracting edges (i.e., removing an edge and merging its two vertices). The *Cartesian product* $G\square H$ of graphs G and H has vertices $V(G\square H) = V(G) \times V(H)$ and an edge between vertices $\langle u, u'\rangle$ and $\langle v, v'\rangle$ iff $u = v$ and $(u', v') \in E(H)$, or $u' = v'$ and $(u, v) \in E(G)$. A (square) *grid* of size n is the Cartesian product of two paths of length n. The *line graph* G_L of a graph G has $V(G_L) = E(G)$, and $(e_1, e_2) \in E(G_L)$ iff edges e_1 and e_2 share a vertex in G.

Let G be a graph, T a rooted tree, and χ a labeling function that maps every node t of T to a subset of $V(G)$ called the *bag* of t. The pair (T, χ) is a *tree decomposition* (Robertson & Seymour, 1986) of G if the following holds: (i) for each $v \in V(G)$, there exists a $t \in T$, such that $v \in \chi(t)$; (ii) for each $\{v, w\} \in E(G)$, there exists a $t \in T$, such that $\{v, w\} \subseteq \chi(t)$; and (iii) for each $r, s, t \in T$, such that s lies on the path from r to t, we have $\chi(r) \cap \chi(t) \subseteq \chi(s)$. The *width* of a tree decomposition is defined as the cardinality of its largest bag minus one. The *treewidth* of a graph G, denoted by $tw(G)$, is the minimum width over all tree decompositions of G. For a minor H of a graph G it holds that $tw(G) \geqslant tw(H)$. Trees have treewidth 1. Grids of size n, the complete graph K_n with n nodes, and the complete

bipartite graph $K_{n,n}$ all have treewidth n. Checking whether a graph has treewidth less than some number k is NP-hard in general, but can be done in linear time for fixed integers k (Bodlaender, 1996). For a more thorough overview of tree decompositions, we refer the reader to standard sources (Bodlaender & Koster, 2010; Niedermeier, 2006; Kloks, 1994).

2.2 Relational Structures

Relational structures are defined over *signatures*. A signature σ is a set of relation symbols, where each symbol R has a non-negative *arity* $\rho_\sigma(R)$. We write $\rho(\sigma)$ to denote the maximum arity of any relation symbol in σ. A *structure* $\mathcal{A}$ over a signature σ consists of a finite *domain* $\mathrm{dom}(\mathcal{A})$ and, for every $R \in \sigma$, a relation $R^{\mathcal{A}} \subseteq \mathrm{dom}(\mathcal{A})^{\rho_\sigma(R)}$. Whenever a relation $R^{\mathcal{A}}$ contains a tuple $\boldsymbol{a}$, we say that $R(\boldsymbol{a})$ is a *fact* in $\mathcal{A}$, and we say that the elements of $\boldsymbol{a}$ are the *arguments* of that fact.

Example 2. For representing directed graphs, it is customary to use the signature $\sigma = \{\mathrm{E}\}$ with $\rho_\sigma(\mathrm{E}) = 2$. We can now represent a directed graph G as a structure $\mathcal{G}$ over σ by choosing $\mathrm{dom}(\mathcal{G}) = \mathrm{V}(G)$ and $\mathrm{E}^{\mathcal{G}} = \mathrm{E}(G)$.

Alternatively, we can also represent G in a slightly more complex way as an "incidence structure": For this, we use the signature $\tau = \{\mathrm{E}, in_1, in_2\}$, where $\rho_\tau(\mathrm{E}) = 1$ and $\rho_\tau(in_1) = \rho_\tau(in_2) = 2$. The intended meaning of the abbreviation "in" is "incident". Now we can define the structure $\mathcal{S}$ over τ via $\mathrm{dom}(\mathcal{S}) = \mathrm{V}(G) \cup \mathrm{E}(G)$, and for each edge e from vertex a to b it holds that $e \in \mathrm{E}^{\mathcal{S}}$, $\langle e, a\rangle \in in_1^{\mathcal{S}}$ and $\langle e, b\rangle \in in_2^{\mathcal{S}}$. △

Generalizing this example, the *incidence structure* of a structure $\mathcal{A}$ is a structure $\mathrm{Inc}(\mathcal{A})$. We call $\mathcal{A}$ the *base structure* of $\mathrm{Inc}(\mathcal{A})$. The signature of $\mathrm{Inc}(\mathcal{A})$ is called the *incidence signature* and denoted $\mathrm{Inc}(\sigma)$, where σ is the signature of $\mathcal{A}$, called the *base signature*. $\mathrm{Inc}(\mathcal{A})$ is defined as follows. Firstly, the domain $\mathrm{dom}(\mathrm{Inc}(\mathcal{A}))$ is the smallest set such that $\mathrm{dom}(\mathcal{A}) \subseteq \mathrm{dom}(\mathrm{Inc}(\mathcal{A}))$ and, for each fact $R(\boldsymbol{a})$ in $\mathcal{A}$, $\langle R, \boldsymbol{a}\rangle \in \mathrm{dom}(\mathrm{Inc}(\mathcal{A}))$. Note that, thus, every fact in $\mathcal{A}$ becomes a domain element in $\mathrm{Inc}(\mathcal{A})$. Secondly, for each relation symbol R in σ, there is a unary relation symbol R in $\mathrm{Inc}(\sigma)$. Finally, there are binary relation symbols $in_0, \ldots, in_k$ in $\mathrm{Inc}(\sigma)$, where $k = \rho(\sigma)$. Now, to complete the definition of $\mathrm{Inc}(\mathcal{A})$, the following facts exist in $\mathrm{Inc}(\mathcal{A})$: (1) for each fact $R(\boldsymbol{a})$ in $\mathcal{A}$, there is a fact $R(\langle R, \boldsymbol{a}\rangle)$ in $\mathrm{Inc}(\mathcal{A})$; and (2) for each fact $R(\boldsymbol{a})$ in $\mathcal{A}$, where a is the i-th argument, $in_i(\langle R, \boldsymbol{a}\rangle, a)$ is a fact in $\mathrm{Inc}(\mathcal{A})$. In this way, the unary relations in $\mathrm{Inc}(\mathcal{A})$ represent the facts from $\mathcal{A}$, and the incidence relations in_i represent the arguments of each fact from $\mathcal{A}$.

In order to apply graph-theoretic concepts (like treewidth) to a structure $\mathcal{A}$, we usually represent $\mathcal{A}$ as its *Gaifman graph*, which is an undirected graph whose vertices are the domain elements of $\mathcal{A}$ and that has an edge between two different elements if they occur together in a fact of $\mathcal{A}$. When we speak of "degree", "distance" or "treewidth" in the context of a structure $\mathcal{A}$, we mean the respective concepts applied to its Gaifman graph.

2.3 MSO Transductions

Monadic second-order (MSO) logic is an extension of first-order logic by quantification over sets. *MSO transductions*, as defined by Courcelle and Engelfriet (2012), use MSO logic to define transformations between structures. Interestingly, they guarantee that, for any fixed

MSO transduction, the treewidth of the *output structure* remains bounded if the treewidth of the *input structure* is bounded as well.

An *MSO definition scheme*, or, simply, *definition scheme*, from a signature σ to a signature σ' is a tuple $\langle \Delta, \Theta \rangle$ where Δ and Θ are sequences of MSO formulas of the following kind, where I denotes a finite set of arbitrary objects:

- For each $i \in I$, Δ contains a formula δ_i with one free variable x. These formulas are called *domain formulas*.
- For each $R' \in \sigma'$ with arity k and $\langle i_1, \dots, i_k \rangle \in I^k$, Θ contains a formula $\theta_{R',i_1,\dots,i_k}$ with k free variables $x_1, \dots, x_k$. These formulas are called *relation formulas*.

The intended meaning of the formulas in a definition scheme $\langle \Delta, \Theta \rangle$ is that they define how to transform an input structure $\mathcal{A}$ into an output structure $\mathcal{A}'$ in the following way: For every element $a \in \mathrm{dom}(\mathcal{A})$ and each formula δ_i such that $\mathcal{A} \vDash \delta_i(a)$, we put a copy of a called (a, i) into $\mathrm{dom}(\mathcal{A}')$. The domain formulas thus define the domain of the output structure. The relation formulas in turn define the relations in the output structure by determining which of the copies of the old domain elements are together in a relation.

A definition scheme $\langle \Delta, \Theta \rangle$ that maps a structure $\mathcal{A}$ over signature σ to a structure $\mathcal{A}'$ over signature σ' is an *MSO transduction* from σ to σ' if it satisfies the following conditions.

- For each $a \in \mathrm{dom}(\mathcal{A})$ and δ_i in Δ, $\mathrm{dom}(\mathcal{A}')$ contains an element (a, i) iff $\mathcal{A} \vDash \delta_i(a)$.
- For each $R' \in \sigma'$ of arity k, all $\delta_{i_1}, \dots, \delta_{i_k}$ occurring in Δ and all $\langle a_1, \dots, a_k \rangle \in \mathrm{dom}(\mathcal{A})^k$, if $\mathcal{A} \vDash \theta_{R',i_1,\dots,i_k}(a_1, \dots, a_k)$ holds in addition to $\mathcal{A} \vDash \delta_{i_j}(a_j)$ for every j, then $R'^{\mathcal{A}'}$ contains a tuple $\langle (a_1, i_1), \dots, (a_k, i_k) \rangle$.

Here, $\mathcal{A}$ is the input structure of the transduction and $\mathcal{A}'$ is the corresponding output structure. Note that the first and second conditions precisely characterize the domain and the relations of $\mathcal{A}'$, respectively.

Thus a (fixed) MSO transduction allows us to copy an input structure a fixed number of times, to filter those domain elements that satisfy a domain formula, and to define the relations of the output structure in terms of the relations of the input structure via the relation formulas. The following examples, taken from Courcelle and Engelfriet (2012), illustrate how such MSO transductions can be used.

Example 3. The following definition scheme formalizes an MSO transduction that transforms a structure $\mathcal{G}$ representing a directed graph into a structure $\mathcal{G}'$ representing the same graph but without self-loops and isolated vertices. We represent directed graphs as structures as in Example 2 using the signature consisting just of the binary relation symbol E.[1]

$$\begin{aligned} \delta_1(x) &\equiv \exists y\big((\mathrm{E}(x,y) \vee \mathrm{E}(y,x)) \wedge x \neq y\big) \\ \theta_{1,1}(x,y) &\equiv \mathrm{E}(x,y) \wedge x \neq y \end{aligned}$$

As there is only one domain formula, we make at most one copy for each vertex in $\mathcal{G}$. In fact, by δ_1, we put a copy $(v, 1)$ into $\mathcal{G}'$ for each vertex v that is adjacent to another vertex.

[1] Since there is only a single relation symbol E in the signature, we write $\theta_{i,j}$ instead of $\theta_{\mathrm{E},i,j}$.

This removes isolated vertices. The relation formula $\theta_{1,1}$ then makes sure that we draw an edge from a copy $(v, 1)$ to a copy $(w, 1)$ if and only if $\mathcal{G}$ contains an edge from v to w and $v \neq w$. △

Example 4. We define an MSO transduction that makes two copies of a directed graph and, for each vertex v with copies $(v, 1)$ and $(v, 2)$, draws an edge from $(v, 1)$ to $(v, 2)$.

$$\begin{aligned} \delta_1(x) &\equiv \delta_2(x) \equiv \top \\ \theta_{1,1}(x, y) &\equiv \theta_{2,2}(x, y) \equiv \mathrm{E}(x, y) \\ \theta_{1,2}(x, y) &\equiv x = y \\ \theta_{2,1}(x, y) &\equiv \bot \end{aligned}$$

The domain formulas unconditionally make two copies of each vertex. The formula $\theta_{1,1}$ then draws an edge from $(v, 1)$ to $(w, 1)$ if there was an edge from v to w, and $\theta_{2,2}$ does the same for the copies with number 2. So far we get two copies of the input graph. Now $\theta_{1,2}$ draws an edge from $(v, 1)$ to $(v, 2)$ for any vertex v. △

As stated earlier, in this paper we will use MSO transductions to investigate the influence of grounding on the treewidth of ASP programs. The following theorem will be an important part of this investigation. It follows straightforwardly from the definition of incidence structures, and Theorem 7.47 in Courcelle and Engelfriet (2012).

Theorem 5. *Let σ and σ' be relational signatures and let f be a function from structures over σ to structures over σ'. If there is an MSO transduction τ corresponding to f in the sense that $\tau(\mathrm{Inc}(\mathcal{A})) = \mathrm{Inc}(f(\mathcal{A}))$ holds for every structure $\mathcal{A}$ over σ, then f preserves bounded treewidth, i.e., for every structure $\mathcal{A}$ over σ, the treewidth of $f(\mathcal{A})$ depends only on the treewidth of $\mathcal{A}$.*

2.4 Answer Set Programming (ASP)

ASP is a declarative problem modeling and solving framework with a complex language that we only briefly introduce here.[2] For a full, formal introduction, we refer to other sources (Gebser et al., 2012; Brewka et al., 2011; Gebser et al., 2012).

An *atom* has the form $p(t_1, \ldots, t_n)$, where p is called a *predicate*. The elements $t_1, \ldots, t_n$ in an atom are called *terms*. A term is either a *constant* or a *variable*. It is customary to write predicates and constants as (strings starting with) lower-case symbols and variables as (strings starting with) upper-case symbols. Moreover, constants may be integers. A *program* in ASP is a set of *rules*, which have the following form:

$$a_1 \vee \ldots \vee a_n \leftarrow b_1, \ldots, b_k, \texttt{ not } b_{k+1}, \ldots, \texttt{not } b_m,$$

where each a_i, $0 < i \leq n$, and b_i, $0 < i \leq m$, is an atom. The *head* of a rule r is the set denoted by $H(r) = \{a_1, \ldots, a_n\}$, the *positive body* of r is the set $B^+(r) = \{b_1, \ldots, b_k\}$, and the *negative body* of r is the set $B^-(r) = \{\texttt{not } b_{k+1}, \ldots, \texttt{not } b_m\}$. The *body* of r is now

[2] We base our definitions on the ASP core language (Calimeri, Faber, Gebser, Ianni, Kaminski, Krennwallner, Leone, Ricca, & Schaub, 2015). However, we omit advanced constructs like aggregates or choice rules, in order to ease presentation.

defined as $B(r) = B^+(r) \cup B^-(r)$. If the head of a rule is empty, then we call the rule a *constraint*. When writing a constraint, we may write $\bot$ before the $\leftarrow$ symbol or we may just omit $\bot$. If the body of a rule is empty, then we we may omit the $\leftarrow$ symbol. If the body of a rule is empty and the head consists of a single (variable-free) atom, then we call the rule a *fact*. A program Π is called *positive* if the negative body of each rule in Π is empty.

We call a program (or rule, or atom) *ground* if it contains no variables. A predicate is called *extensional* in a program Π if it only occurs in rule bodies of Π. A *literal* is an atom a or its negated form `not` a. A rule r is *safe* if each variable that occurs in the head or negative body of r also occurs in its positive body. A program is safe if all its rules are safe. We only admit ASP programs that are safe.

Semantics. We define the semantics of ASP in terms of ground programs. To this end, let the *Herbrand universe* of a program Π, denoted by U_Π, be the set of all constants occurring in Π. The *Herbrand base* of Π is the set of all atoms that can be constructed using predicates occurring in Π and constants in U_Π. We denote this set by B_Π. Subsets of B_Π are called *interpretations* and intuitively correspond to the atoms that we set to true while all other atoms are set to false.

We now define the *ground instantiation* of a program Π as the set of all ground instances of all rules in Π, that is, the set of ground rules obtained from the rules in Π by replacing all variables by all different combinations of constants from the Herbrand universe U_Π. However, modern ASP grounders perform many optimizations and do not blindly ground programs in the described way. In order for our investigation in this paper to be meaningful, we assume that the following optimization, present in all reasonable ASP grounder implementations today, is made: let Π^+ denote the positive program obtained from Π by removing the negative bodies of all rules and replacing disjunctions in the heads with conjunctions[3]. We say that an atom is *possibly true in* Π iff it is contained in the unique minimal model of Π^+. We will use the following idealized definition of grounding for the remainder of our paper.

Definition 6. The *grounding* of an ASP program Π, denoted $\mathrm{gr}(\Pi)$, is the set of all those rules r in the ground instantiation of Π where every atom in $B^+(r)$ is possibly true.

Given an interpretation I of a program Π, we say that a ground atom a is *true* under I if $a \in I$, otherwise it is *false*. Similarly, we say that a negated ground atom a, written as `not` a, is *true* under I if $a \notin I$, otherwise it is *false*. An interpretation I of a program Π *satisfies* a rule r in $\mathrm{gr}(\Pi)$ if $B(r)$ being true under I implies $H(r)$ being true under I. We say that I is a *model* of Π if it satisfies every rule in $\mathrm{gr}(\Pi)$.

The central notion for ASP is that of the *answer set*. Also called *stable model*, this was originally introduced by Gelfond and Lifschitz (1988) and later extended to disjunctive programs by (Gelfond & Lifschitz, 1991) (1991). In order to define answer sets, we first define the *reduct* of a program Π w.r.t. an interpretation I, denoted Π^I, as the set of those rules in $\mathrm{gr}(\Pi)$ whose body elements are all true under I. Now I is an *answer set* of Π if I is a model of Π and no proper subset of I is a model of Π^I. Given an answer set I and a

[3] That is, we replace a rule r whose head is $h_1 \vee \cdots \vee h_k$ by rules $r_1, \ldots, r_k$ such that $H(r_i) = \{h_i\}$ and the body of r_i is $B^+(r)$.

predicate p, we say that the set $\{\langle t_1, \ldots, t_n\rangle \mid p(t_1, \ldots, t_n) \in I\}$ is the *extension* of p under I.

Complexity. In this work, we are mainly interested in the complexity of the Answer Set Existence decision problem, defined below.

> Answer Set Existence
> Input: An ASP program Π
> Question: Does Π have an answer set?

If the program is ground, this problem is at the second level of the polynomial hierarchy, namely complete for Σ_2^{P} (Eiter & Gottlob, 1995). We are particularly interested in the *data complexity* of Answer Set Existence. The term data complexity comes from the study of query languages, where we are faced with problem instances consisting of a query and data, and the question is whether the query holds on the data (or, more generally, to evaluate the query on the data). ASP fits well into this framework, since it is commonly used for encoding a problem as a fixed non-ground program and then combining this fixed encoding with ground facts that represent an actual input instance of that problem. Hence, the Answer Set Existence problem can be reformulated as follows:

> Answer Set Existence
> Input: A non-ground ASP program Π, and a set of facts F
> Question: Does $\Pi \cup F$ have an answer set?

Eiter, Gottlob, and Mannila (1997) showed that for fixed programs Π it remains Σ_2^{P}-complete to decide whether $\Pi \cup F$ has an answer set for some set of input facts F. This can be seen from the fact that the size of $\mathrm{gr}(\Pi \cup F)$ remains polynomial in the size of F.

2.5 Treewidth of Ground ASP Programs

We can easily apply the parameter treewidth to ground ASP programs by defining a suitable representation as a graph. The *primal graph* of a ground ASP program Π is a graph whose vertices are the atoms in Π and there is an edge (a, b) if atoms a and b appear together in a rule in Π. The *incidence graph* of Π is a bipartite graph whose vertices are the atoms and rules in Π and there is an edge between a rule r and an atom a iff a appears in r. When we speak of the *treewidth of a ground ASP program*, we usually refer to the treewidth of its primal graph.

On ground ASP programs, the Answer Set Existence problem parameterized by the treewidth of the primal graph is fixed-parameter tractable (Gottlob et al., 2010). In fact, it can even be solved in linear time when the treewidth is bounded by a constant.

Example 7. Consider again the transitivity-based reachability encoding from Example 1:

```
t(X,Y) :- e(X,Y).
t(X,Z) :- t(X,Y), e(Y,Z).
r(Y) :- t(X,Y), start(X).
```

Let C be the class of graphs consisting of all chains of finite length. Each element of C has treewidth 1, hence C is a class of graphs of bounded treewidth. Grounding the reachability encoding together with elements of C as input (using the straightforward encoding as facts), however, leads to ground programs with unbounded treewidth. Indeed, if we ground it with a chain involving vertices $1, \ldots, n$, then the primal graph of the grounding will contain an edge between $\mathtt{t}(i,j)$ and $\mathtt{t}(i,k)$, for each $i, j, k \in \{1, \ldots, n\}$, because these atoms occur together in a ground instantiation of the second rule. The grounding for this chain thus contains K_n as a subgraph and therefore has treewidth n. Hence, even though the class of input graphs has bounded treewidth, the groundings have unbounded treewidth. △

2.6 Input Facts as Relational Structures

We will, throughout this paper, often see sets of ASP facts as relational structures, relative to some (non-ground) program Π. To do this formally, we choose the extensional predicates in Π as the signature and interpret each such predicate as its extension in the facts. In addition, since ASP systems always assume that the Herbrand base is ordered (in an arbitrary way), we can assume that the signature also contains a binary successor relation denoted by *succ*, which is interpreted according to an arbitrary order of the domain elements.

Let Π be an ASP program whose set of extensional predicates we denote by τ. For every set F of input facts for Π, we define the *fact structure* of F to be the structure $\mathcal{F}$ over $\tau \cup \{succ\}$ where $\mathrm{dom}(\mathcal{F})$ consists of all ASP constants occurring in F, and for every predicate $p \in \tau$ it holds that $p^{\mathcal{F}}$ is the set of all k-tuples $\boldsymbol{a}$ such that $p(\boldsymbol{a})$ is a fact in F. We interpret the *succ* relation as an arbitrary, fixed successor relation of the ASP constants. By slight abuse of notation, we sometimes write $\mathcal{F}$ in place of F. For instance, we may write $\mathrm{gr}(\Pi \cup \mathcal{F})$ instead of $\mathrm{gr}(\Pi \cup F)$ to denote the ground instantiation of Π together with the input facts F. The *input structures* of a program Π comprise the fact structures of all sets of input facts for Π.

Example 8. Let Π be an ASP program whose only extensional predicate is p, and let F be the set of input facts for Π consisting of $p(a,b)$ and $p(a,c)$. The fact structure $\mathcal{F}$ of F contains the domain elements a, b and c, and it interprets the binary relation symbol p as $\{\langle a,b\rangle, \langle a,c\rangle\}$. Moreover, it contains an arbitrary interpretation of the *succ* relation, for instance $succ^{\mathcal{F}} = \{\langle a,b\rangle, \langle b,c\rangle\}$. △

Sometimes we are also interested in the incidence structure of a fact structure. For illustration purposes, this is done in the following example.

Example 9. Continuing Example 8, we denote the facts $\langle a,b\rangle$ and $\langle a,c\rangle$ in $p^{\mathcal{F}}$ by ab_p and ac_p, respectively, and we denote the facts $\langle a,b\rangle$ and $\langle b,c\rangle$ in $succ^{\mathcal{F}}$ by ab_{succ} and bc_{succ}, respectively. Now the domain of $\mathrm{Inc}(\mathcal{F})$ is $\{a, b, c, ab_p, ac_p, ab_{succ}, bc_{succ}\}$. The relations are interpreted as follows:

$$
\begin{aligned}
p^{\mathrm{Inc}(\mathcal{F})} &= \{ab_p, ac_p\} \\
succ^{\mathrm{Inc}(\mathcal{F})} &= \{ab_{succ}, bc_{succ}\} \\
in_1^{\mathrm{Inc}(\mathcal{F})} &= \{\langle ab_p, a\rangle, \langle ac_p, a\rangle, \langle ab_{succ}, a\rangle, \langle bc_{succ}, b\rangle\} \\
in_2^{\mathrm{Inc}(\mathcal{F})} &= \{\langle ab_p, b\rangle, \langle ac_p, c\rangle, \langle ab_{succ}, b\rangle, \langle bc_{succ}, c\rangle\} \triangle
\end{aligned}
$$

```
1 assign(X,Y,1) | assign(X,Y,0) :- edge(X,Y).
2 sum(V,S\2) :- vertex(V), S = #sum{ A,W : assign(W,V,A); A,W :
      assign(V,W,A) }.
3 :- sum(V,S), capacity(V,C), S != C.
```

Listing 1: Finding valid assignments in capacitated graphs.

3. Impact of Treewidth on ASP Solvers

In this section, our goal is to establish that current ASP solvers are impacted by the treewidth of a given ground input ASP program. We will demonstrate, by experimental evaluation, that state-of-the-art systems have an inherent sensitivity to treewidth in practice, and hence perform faster on ground programs of small treewidth. To show this claim, a carefully designed experiment is needed in order to actually reveal the influence of the treewidth (and not some other parameter) on the solving time. We therefore need to fix several other parameters in order to exclude them from influencing the runtime of the ASP system throughout our benchmarks. In particular, we want to establish four conditions. The ground programs used for this experiment should

(1) have the same number of answer sets,
(2) have a uniform structure,
(3) have constant size, and
(4) vary in the treewidth.

To this end, we consider the problem of deciding whether a capacitated graph has a valid assignment:

Definition 10. A *capacitated graph* is a pair (G, c), where G is an undirected graph and c is a function mapping each vertex to 0 or 1. A function mapping each edge to 0 or 1 is called an *assignment*, and we call it *valid* if for each $v \in V(G)$ the sum modulo 2 of the values assigned to incident edges equals $c(v)$. □

This problem was used by Urquhart to construct hard SAT formulas (Urquhart, 1987) using the method of Tseitin (1983). Formally, it is defined as follows:

Valid Assignment Existence	
Input:	A capacitated graph (G, c)
Question:	Does a valid assignment exist for (G, c)?

Listing 1 shows an ASP encoding for this problem[4]. Line 1 assigns either 0 or 1 to each edge. Line 2 calculates for each vertex the sum modulo 2 of the values assigned to incident edges (the backslash represents the modulo operation in ASP). Finally, Line 3 is

[4] Note that in this problem encoding, we use some advanced constructs of the ASP-Core-2 language not introduced in Section 2. However, we do explain the meaning in the main text, and advanced constructs like aggregates are removed by the grounding process. For exact semantic definitions, we refer the reader to Calimeri et al. (2015) and Gebser, Kaminski, Kaufmann, Lindauer, Ostrowski, Romero, Schaub, and Thiele (2015).

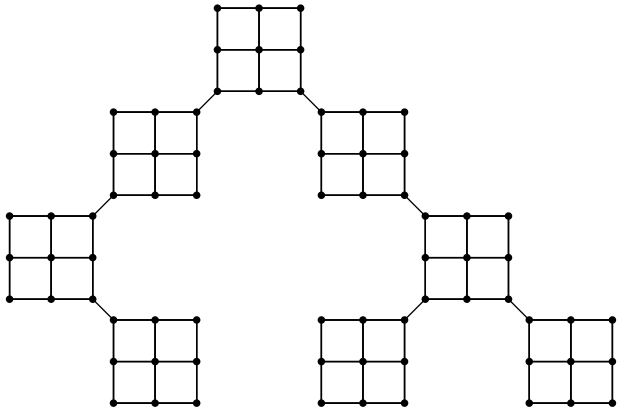


Figure 1: Example for an input instance graph.

a constraint that eliminates all assignments where for some vertex the sum and capacity do not agree. We will now show how we can generate instances that fulfill all four of our conditions and are thus usable for our experiment.

Input Instances. In order to satisfy condition (1), we will only construct unsatisfiable instances. It is known that a connected capacitated graph has a valid assignment iff the sum of all vertex capacities is even (Urquhart, 1987). To construct unsatisfiable instances, we thus generate graphs where all vertices except for one have capacity 0.

To satisfy condition (2), we choose Listing 1 as our fixed problem encoding and construct instances in the following way. We generate so-called *grid-tree* instances, which are random binary trees of grids as illustrated in Figure 1, according to two parameters: *treesize* (the number of grids in the tree) and *gridsize* (the size of each grid).

Given a grid-tree instance, we can find other grid-tree instances that lead to the same grounding size by increasing the treesize while decreasing the gridsize (or vice versa). Thus, in order to satisfy condition (3), we first fix a grounding size (in terms of the number of atoms), and then find combinations of treesize and gridsize to achieve this grounding size.

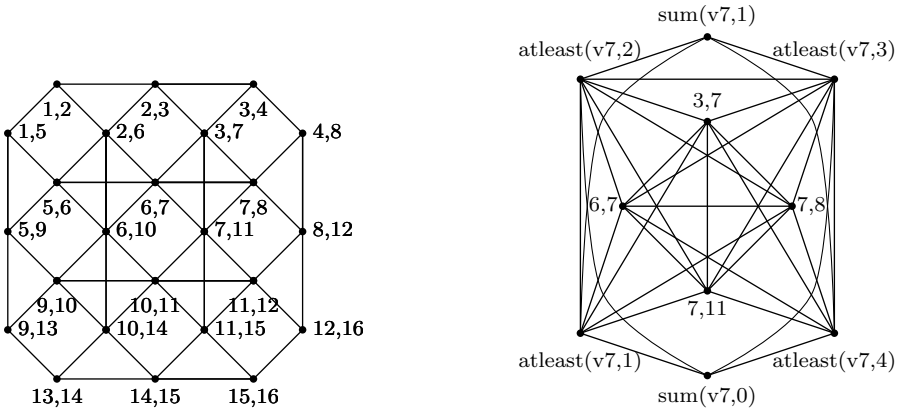
Finally, we need to establish condition (4). For a (non-empy) graph G consisting only of a disjoint union of grids $G_1, \ldots, G_k$ plus some edges that do not create any new cycles, it holds that $tw(G) \leq \max\{tw(G_1), \ldots, tw(G_k)\}$, since we can easily obtain an appropriate tree decomposition of G from those of $G_1, \ldots, G_k$. Hence the treewidth of a grid-tree instance $\mathcal{A}$ only depends on the gridsize. We now need to show that the treewidth of $\mathcal{A}$ determines the treewidth of the grounding.

Proposition 11. *Given program* Π *from Listing 1 and a grid-tree instance* $\mathcal{A}$*, the treewidths of the primal and incidence graphs of* $\mathrm{gr}(\Pi \cup \mathcal{A})$ *are both linear in the treewidth of* $\mathcal{A}$*.*

Assume, for simplicity, that $\mathcal{A}$ is just a single grid of size 4. We call the nodes in the first row of the grid `v1`, ..., `v4`, the nodes in the second row `v5`, ..., `v8`, and so on. In order to show the above proposition for $\mathcal{A}$, first note that Line 1 and Line 3 in Listing 1 cannot cause any cycles in the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{A})$. Now consider Listing 2, which shows the grounding for Line 2 of Listing 1 for vertex `v7` of $\mathcal{A}$. Figure 2b shows the primal graph of this partial grounding (where `M,N` represents the atom `assign(vM,vN,1)`). Clearly, the rule

```
1 atleast(v7,1) :- 1 { assign(v3,v7,1), assign(v6,v7,1), assign(v7,v8,1),
      assign(v7,v11,1) }.
2 atleast(v7,2) :- 2 { assign(v3,v7,1), assign(v6,v7,1), assign(v7,v8,1),
      assign(v7,v11,1) }.
3 atleast(v7,3) :- 3 { assign(v3,v7,1), assign(v6,v7,1), assign(v7,v8,1),
      assign(v7,v11,1) }.
4 atleast(v7,4) :- 4 { assign(v3,v7,1), assign(v6,v7,1), assign(v7,v8,1),
      assign(v7,v11,1) }.
5 sum(v7,0) :- not atleast(v7,1).
6 sum(v7,0) :- atleast(v7,2), not atleast(v7,3).
7 sum(v7,0) :- atleast(v7,4).
8 sum(v7,1) :- atleast(v7,1), not atleast(v7,2).
9 sum(v7,1) :- atleast(v7,3), not atleast(v7,4).
```

Listing 2: Grounding of Line 2 in Listing 1 for vertex v7.



(a) The line graph of a grid of size 4.

(b) Primal graph of Listing 2.

Figure 2: Structure of the primal graph of Listing 2.

bodies of the first four lines in Listing 2 cause the central clique between vertices $3, 7$; $7, 8$; $7, 11$; and $6, 7$ to appear. Note that this forms a clique precisely between the incident edges of vertex v7. Now, take only the rule bodies of Lines 1 to 4 in Listing 2, and the same also from the analogous groundings of every other vertex in $\mathcal{A}$. The corresponding partial primal graph is shown in Figure 2a. In fact, this is precisely the line graph of the grid in $\mathcal{A}$. The full primal graph of $\mathrm{gr}(\Pi \cup \mathcal{A})$ can now be obtained by replacing every clique in Figure 2a (which represents some vertex v from $\mathcal{A}$) with a gadget analogous to the one in Figure 2b, but for vertex v. Note that this gadget has constant size (and, therefore, treewidth). We thus have that the treewidth of $\mathrm{gr}(\Pi \cup \mathcal{A})$ is asymptotically upper-bounded by the treewidth of the line graph of $\mathcal{A}$. It is not difficult to extend this argument to the case where $\mathcal{A}$ is a grid-tree instead of a single grid.

To complete our line of argument, note that it is known that the treewidth of the line graph is linear in the treewidth of the original graph if the maximum degree of the latter is bounded by a constant, as is the case for grids (Călinescu, Fernandes, & Reed, 2003). Since the treewidth of the incidence graph is upper-bounded by the treewidth of the primal graph (Szeider, 2003), this establishes condition (4).

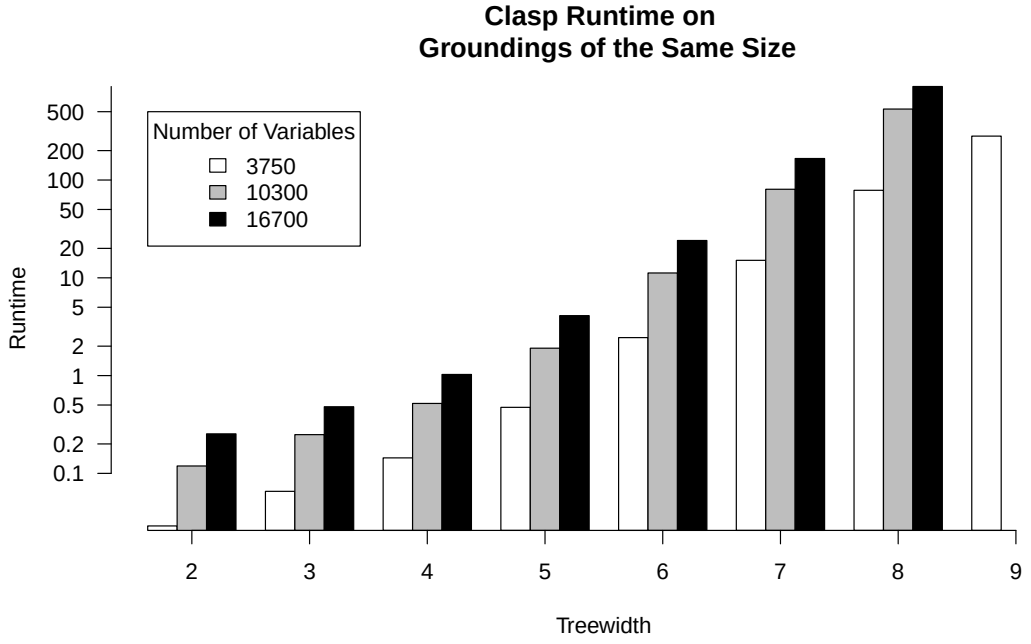


Figure 3: Results for clasp.

Benchmark Setting. In our tests we used three batches of instances constructed as presented before, each batch for a different grounding size. The size of the ground program was determined by the number of variables reported by *clasp*. For the first batch the number of variables in the grounding is approximately 3750, for the second it is 10300, and for the third 16700, where each instance is allowed to deviate up to ± 100 variables. In each batch we generated 20 instances for each treewidth between 2 and 9 in order to account for fluctuations in the running time.[5] Grounding was done using the grounder *gringo* 4.5.4 (Gebser, Kaufmann, Kaminski, Ostrowski, Schaub, & Schneider, 2011). We then measured the running time of the ASP solvers *clasp* 3.1.4 (Gebser et al., 2011) and *WASP* 2.0 (Alviano et al., 2015).

Results and Discussion. Figures 3 and 4 show how the average solving time of *clasp* and *WASP* changes with the treewidth for our three batches of instances. Recall that the treewidth of the corresponding ground program is linear in the input treewidth by Proposition 11. Since the running time in the figures is plotted on a logarithmic scale, but, starting at treewidth 4, seems to form a line, it seems that the running time increases exponentially with the treewidth while the grounding size remains constant. Moreover, the running time for small programs with high treewidth can be substantially longer than for large programs with small treewidth.

From these benchmark results, we thus conclude that the treewidth of the grounding has a major impact on the running time of current ASP solvers and that, for good solving

[5] Full archive: `http://dbai.tuwien.ac.at/proj/decodyn/ijcai17-benchmarks.zip`

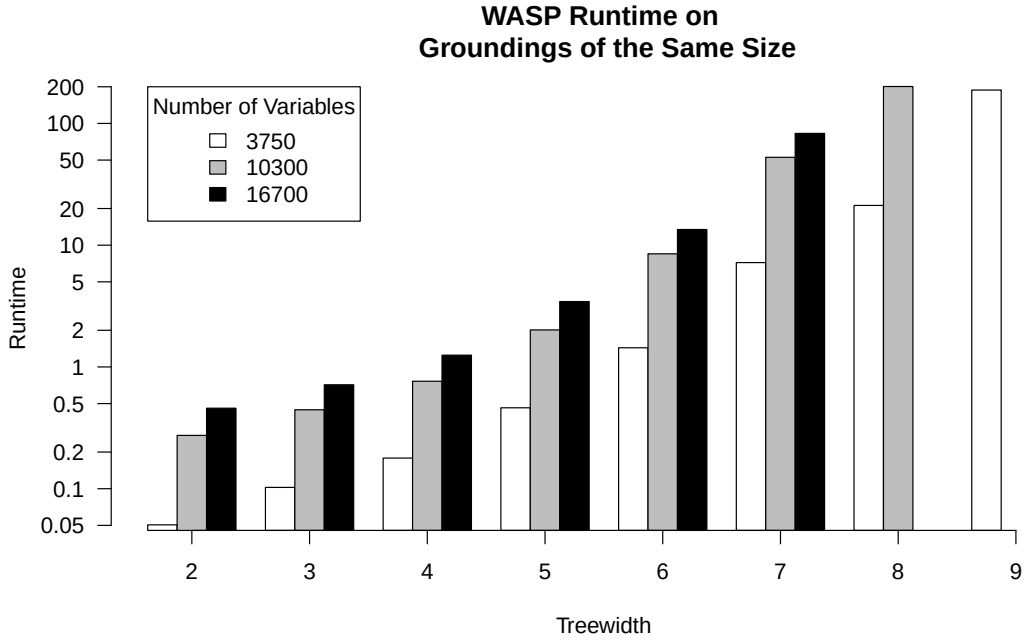


Figure 4: Results for WASP.

performance, it is important to keep the treewidth of ground programs as low as possible. The following observation formalizes this.

Observation 12. *For programs of constant size but varying treewidth and current CDCL-based ASP solvers, it holds that higher treewidth implies exponentially higher solving time.*

4. Treewidth-Preserving Classes of Programs

As Observation 12 shows us, the influence of the treewidth of an ASP program on the solving time with today's state-of-the-art ASP solvers is quite substantial. Ground ASP programs can be solved much faster if their treewidth is low. This begs the question of what can be done to formulate ASP programs in such a way as to keep the treewidth as low as possible.

Recall that ASP programs are not usually written directly in their ground form, but are written as a (usually fixed) non-ground ASP encoding for a given problem. This encoding, together with a problem instance encoded as ground ASP facts, is then grounded using an ASP grounder. In this section, we will therefore investigate how encodings can be written in such a way that, after grounding, the structure present in the ground facts (i.e., the problem instance) is preserved as best as possible.

To this end, we will study two classes of non-ground ASP programs that guarantee that the ground ASP program obtained from a set of facts and an encoding that falls into the class has bounded treewidth if the set of facts already has certain structural properties, specified below. In the following let Π be a non-ground ASP program, and $\mathcal{A}$ a set of input facts.

Guarded ASP Programs: For programs Π in this class, the treewidth of a ground program $\mathrm{gr}(\Pi \cup \mathcal{A})$ depends only on the treewidth of the fact structure of $\mathcal{A}$.

Connection-Guarded ASP Programs: For programs Π in this class, the treewidth of the ground program $\mathrm{gr}(\Pi \cup \mathcal{A})$ depends only on the treewidth and degree of the fact structure of $\mathcal{A}$.

This section defines these classes. Following these definitions, we formally show that the properties claimed above indeed hold for ground programs obtained from non-ground ASP encodings that fall into one of our two classes. In addition, we discuss why our classes are, in a certain sense, "optimal" in that any restrictions they impose are actually needed to guarantee those properties.

In Section 5, we will then investigate computational properties of our classes. In particular, we will show that guarded programs, despite a rather restrictive syntax, are still expressive enough to encode relevant problems on the second level of the polynomial hierarchy. Since every guarded program is connection-guarded, this also holds for connection-guarded ASP, which offers a richer syntax.

The remainder of this section is structured as follows: Section 4.1 provides an overview of our techniques (i.e., our "plan of attack") and lays out some basic definitions and properties needed later to define our two classes. Then, in Section 4.2, we define the class of guarded ASP programs and prove that grounding fixed programs in this class preserves bounded treewidth of the input. In Section 4.3, this is then followed by our definition of connection-guarded ASP programs and the proof that grounding fixed programs in this class preserves bounded treewidth of the input if additionally the degree of the input is bounded.

4.1 Overview

In order to define and investigate our two classes of ASP programs, we will make use of MSO logic to express properties of arbitrary structures, in order to formalize the grounding process for our classes as MSO transductions. This will allow us to establish the relevant properties described above.

Let us first introduce some notation, which will apply throughout the remainder of this section. Let Π be some fixed, non-ground ASP program that falls within one of our classes, and let $\mathcal{A}$ be a fact structure representing the input instance consisting of a set of facts. We will denote the signature of $\mathcal{A}$ (and Π) with σ (also called the base signature). As stated in Section 2, we will assume that every such signature contains a binary successor relation predicate *succ* that specifies an order over the domain elements in $\mathcal{A}$. This assumption is natural, since ASP solvers and grounders, in practice, always assume an order over the constants in a program. We will also assume that $\mathcal{A}$ has the same signature as Π. Therefore, note that σ is fixed, since so is Π. Finally, let $\rho_{\max}$ be the maximum arity of any predicate within σ (i.e., $\rho_{\max} = \rho(\sigma)$).

Now, in order to show our results, our plan of attack is as follows. Each non-ground program Π gives rise to an MSO transduction γ that simulates the grounding process for Π. These MSO transductions will always start from the incidence structure $\mathrm{Inc}(\mathcal{A})$, which serves as the input to our MSO formulas. The output of the MSO transduction, that is $\gamma(\mathrm{Inc}(\mathcal{A}))$, will be the incidence structure of the primal graph of the ground program $\mathrm{gr}(\Pi \cup \mathcal{A})$. Recall

that this primal graph contains the atoms in $\mathrm{gr}(\Pi \cup \mathcal{A})$ as vertices, and an edge between them if they appear together in a rule in $\mathrm{gr}(\Pi \cup \mathcal{A})$. Hence, our MSO transductions must be able to identify the atoms that appear in $\mathrm{gr}(\Pi \cup \mathcal{A})$, and, more importantly, only generate exactly one vertex per such atom. Hence, before giving the proofs of our main theorems, we will define MSO subformulas that we will later use to uniquely identify an atom that appears in $\mathrm{gr}(\Pi \cup \mathcal{A})$ based on a lexicographic order of the domain elements in $\mathcal{A}$, given by the *succ* predicate. In our MSO formulas, we will use the symbol $<$ as shorthand for the strict total order on the domain elements of $\mathcal{A}$ that naturally arises from the *succ* relation[6].

4.2 Guarded Answer Set Programs

In this section, we define the class of guarded ASP programs and show that they lead to groundings whose treewidth depends only on the treewidth of the input structure.

Definition 13. Let Π be an ASP program. A *guard* of a rule r in Π is an extensional atom in the positive body of r that contains every variable occurring in r. We call Π *guarded* if every rule has a guard. We designate an arbitrary guard of r as *the guard* of r.

The objective of this section is to establish the following, central theorem about guarded ASP programs.

Theorem 14. *Let Π be a fixed guarded ASP program, and $\mathcal{A}$ be an input structure of Π. If $\mathcal{A}$ has bounded treewidth, then the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{A})$ has bounded treewidth.*

Note that this theorem immediately implies that the treewidth of the primal graph depends only on the treewidth of input structure $\mathcal{A}$. This ensures that ASP programs that are written in such a way that they are guarded ensure that ASP solvers will be able to solve them efficiently, in accordance with Observation 12 in Section 3.

In order to prove this main theorem, we need to introduce several preliminary notions first. The formulas defined below will allow us to identify certain tuples $\langle a_1, \ldots, a_k \rangle$ of domain elements by a fact and a tuple of integers $\langle i_1, \ldots, i_k \rangle$ such that each a_j is the i_j-th argument of that fact. Note that this is the general idea underlying the class of guarded ASP programs: only if some domain elements already occur together in a fact of $\mathcal{A}$ can they give rise to a new ground atom in $\mathrm{gr}(\Pi \cup \mathcal{A})$.

We begin with the formula $\mathrm{fact}(x)$, which is true under a structure $\mathrm{Inc}(\mathcal{A})$ if x is a fact in the base structure $\mathcal{A}$. Recall that the domain of $\mathrm{Inc}(\mathcal{A})$ consists of the domain of $\mathcal{A}$ and the facts in $\mathcal{A}$, and that the relation symbols in the base signature σ are all unary in $\mathrm{Inc}(\sigma)$. Recall that all subformulas presented in this section are formulated to operate on the incidence signature $\mathrm{Inc}(\sigma)$.

$$\mathrm{fact}(x) \equiv \bigvee_{R \in \sigma} R(x)$$

[6] We can easily define $<$ from the *succ* relation because transitive closure can be defined in MSO (Courcelle & Engelfriet, 2012, Section 1.3.1). Moreover, note that *succ* only gives us an ordering on the ASP constants in $\mathcal{A}$, but not on the facts. However, these are also present as domain elements of $\mathrm{Inc}(\mathcal{A})$, which we use as the input to our MSO transductions. This is not a problem, however, since we can define a lexicographical order on tuples of constants based on the ordering on the individual constants.

The following formula expresses that x is a fact that contains y as some argument.

$$in(x, y) \;\equiv\; in_1(x, y) \vee \cdots \vee in_{\rho_{\max}}(x, y)$$

If a fact in a base structure $\mathcal{A}$ contains every element of a tuple $\boldsymbol{a}$ of elements of $\mathrm{dom}(\mathcal{A})$, then we say that this fact *covers* $\boldsymbol{a}$. For each nonnegative integer k, we define the following formula, which states that x is a fact that covers $\langle y_1, \ldots, y_k \rangle$.

$$\mathrm{covers}_k(x, y_1, \ldots, y_k) \;\equiv\; \mathrm{fact}(x) \wedge in(x, y_1) \wedge \cdots \wedge in(x, y_k)$$

Furthermore, if some fact of a base structure $\mathcal{A}$ covers $\boldsymbol{a}$, then we define the *first cover* of $\boldsymbol{a}$ in $\mathcal{A}$ to be the smallest fact covering $\boldsymbol{a}$ according to the order of the domain elements, which we can construct from the successor relation that is guaranteed to be present in the input structure. For each nonnegative integer k, we therefore define a formula to express that x is the first cover of $\langle y_1, \ldots, y_k \rangle$:

$$\mathrm{fcov}_k(x, y_1, \ldots, y_k) \;\equiv\; \mathrm{covers}_k(x, y_1, \ldots, y_k) \wedge \neg\exists z\big(z < x \wedge \mathrm{covers}_k(z, y_1, \ldots, y_k)\big)$$

We say that a tuple $\langle i_1, \ldots, i_k \rangle$ of integers *extracts* a tuple $\langle a_1, \ldots, a_k \rangle$ of domain elements from a fact $R(d_1, \ldots, d_\ell)$ if both $1 \leq i_j \leq \ell$ and $a_j = d_{i_j}$ hold for every j. Clearly there is a tuple of integers that extracts a tuple $\boldsymbol{a}$ of domain elements from a fact x if and only if x covers $\boldsymbol{a}$. For each nonnegative integer k and every tuple $\langle i_1, \ldots, i_k \rangle$ of integers between 1 and $\rho_{\max}$, we define the following formula to express that $\langle i_1, \ldots, i_k \rangle$ extracts $\langle y_1, \ldots, y_k \rangle$ from x:

$$\mathrm{extract}_{\langle i_1, \ldots, i_k \rangle}(x, y_1, \ldots, y_k) \;\equiv\; in_{i_1}(x, y_1) \wedge \cdots \wedge in_{i_k}(x, y_k)$$

If a fact x covers $\boldsymbol{a}$, then we define the *first tuple* extracting $\boldsymbol{a}$ from x as the lexicographically smallest tuple of integers between 1 and $\rho_{\max}$ that extracts $\boldsymbol{a}$ from x. For this we first define the relation $\prec_{\rho_{\max}}$ among tuples of integers between 1 and $\rho_{\max}$ such that $a \prec_{\rho_{\max}} b$ holds if a is lexicographically smaller than b. For every nonnegative integer k and every k-tuple $\boldsymbol{i}$ of integers between 1 and $\rho_{\max}$, we now define the following formula, which is true if and only if $\boldsymbol{i}$ is the first tuple extracting $\langle y_1, \ldots, y_k \rangle$ from x:

$$\mathrm{fext}_{\boldsymbol{i}}(x, y_1, \ldots, y_k) \;\equiv\; \mathrm{extract}_{\boldsymbol{i}}(x, y_1, \ldots, y_k) \wedge \bigwedge_{\boldsymbol{j} \prec_{\rho_{\max}} \boldsymbol{i}} \neg\, \mathrm{extract}_{\boldsymbol{j}}(x, y_1, \ldots, y_k)$$

If a fact in a base structure $\mathcal{A}$ covers a tuple $\boldsymbol{a}$ of domain elements, we define the *cover-based identifier* of $\boldsymbol{a}$ in $\mathcal{A}$ to be the unique combination of a fact x and a tuple $\boldsymbol{i}$ of integers such that x is the first cover of $\boldsymbol{a}$ in $\mathcal{A}$ and $\boldsymbol{i}$ is the first tuple extracting $\boldsymbol{a}$ from x. For each nonnegative integer k and every k-tuple $\boldsymbol{i}$ of integers between 1 and $\rho_{\max}$, we now define the following formula to express that x together with $\boldsymbol{i}$ is the cover-based identifier of $\langle y_1, \ldots, y_k \rangle$:

$$\mathrm{cid}_{\boldsymbol{i}}(x, y_1, \ldots, y_k) \;\equiv\; \mathrm{fcov}_k(x, y_1, \ldots, y_k) \wedge \mathrm{fext}_{\boldsymbol{i}}(x, y_1, \ldots, y_k)$$

Whenever a fact covers $\boldsymbol{a}$, the cover-based identifier of $\boldsymbol{a}$ clearly exists because then $\boldsymbol{a}$ has a first cover and $\boldsymbol{a}$ can be extracted from this cover. For every fact x and each tuple $\boldsymbol{i}$

of integers, the combination of x and $\boldsymbol{i}$ is the cover-based identifier of at most one tuple of domain elements. Hence there is a bijection between the set of all tuples that have a cover-based identifier and the set of all cover-based identifiers. Note that, in particular, every tuple of constants that appears in an atom of $\mathrm{gr}(\Pi \cup \mathcal{A})$ has exactly one cover-based identifier.

Example 15. Let $\mathcal{A}$ be a structure over a signature consisting of the binary relation symbols R and *succ* such that $\mathrm{dom}(\mathcal{A}) = \{a, b\}$, $R^{\mathcal{A}} = \{\langle a, a\rangle\}$ and $succ^{\mathcal{A}} = \{\langle a, b\rangle\}$. We denote the domain elements of $\mathrm{Inc}(\mathcal{A})$ corresponding to the facts $\langle a, a\rangle$ in $R^{\mathcal{A}}$ and $\langle a, b\rangle$ in $succ^{\mathcal{A}}$ by aa and ab, respectively. We assume that the ordering of the domain elements of $\mathrm{Inc}(\mathcal{A})$ that we extracted from $succ^{\mathcal{A}}$ is $a < b < aa < ab$. The following formulas are true under $\mathrm{Inc}(\mathcal{A})$:

- $\mathrm{fact}(aa)$, $\mathrm{fact}(ab)$.
- $in(aa, a)$, $in(ab, a)$, $in(ab, b)$.
- $\mathrm{covers}_1(aa, a)$, $\mathrm{covers}_2(aa, a, a)$, $\mathrm{covers}_3(aa, a, a, a)$, ...

  $\mathrm{covers}_1(ab, a)$, $\mathrm{covers}_1(ab, b)$. $\mathrm{covers}_2(ab, a, a)$, $\mathrm{covers}_2(ab, a, b)$, $\mathrm{covers}_2(ab, b, a)$, $\mathrm{covers}_2(ab, b, b)$. $\mathrm{covers}_3(ab, a, a, a)$, ...
- $\mathrm{fcov}_1(aa, a)$, $\mathrm{fcov}_2(aa, a, a)$, $\mathrm{fcov}_3(aa, a, a, a)$, ...

  $\mathrm{fcov}_1(ab, b)$. $\mathrm{fcov}_2(ab, a, b)$, $\mathrm{fcov}_2(ab, b, a)$, $\mathrm{fcov}_2(ab, b, b)$, $\mathrm{fcov}_3(ab, a, a, b)$, ...

  But note that, e.g., $\mathrm{fcov}_1(ab, a)$ is not true even though ab covers a, since a is also covered by aa, which is less than ab.
- $\mathrm{extract}_{\langle 1\rangle}(aa, a)$, $\mathrm{extract}_{\langle 2\rangle}(aa, a)$, $\mathrm{extract}_{\langle 1,1\rangle}(aa, a, a)$, $\mathrm{extract}_{\langle 1,2\rangle}(aa, a, a)$, $\mathrm{extract}_{\langle 2,1\rangle}(aa, a, a)$, $\mathrm{extract}_{\langle 2,2\rangle}(aa, a, a)$, $\mathrm{extract}_{\langle 1,1,1\rangle}(aa, a, a, a)$, ...

  $\mathrm{extract}_{\langle 1\rangle}(ab, a)$, $\mathrm{extract}_{\langle 2\rangle}(ab, b)$, $\mathrm{extract}_{\langle 1,1\rangle}(ab, a, a)$, $\mathrm{extract}_{\langle 1,2\rangle}(ab, a, b)$, $\mathrm{extract}_{\langle 2,1\rangle}(ab, b, a)$, $\mathrm{extract}_{\langle 2,2\rangle}(ab, b, b)$, $\mathrm{extract}_{\langle 1,1,1\rangle}(ab, a, a, a)$, ...
- $\mathrm{fext}_{\langle 1\rangle}(aa, a)$, $\mathrm{fext}_{\langle 1,1\rangle}(aa, a, a)$, $\mathrm{fext}_{\langle 1,1,1\rangle}(aa, a, a, a)$, ...

  $\mathrm{fext}_{\langle 1\rangle}(ab, a)$, $\mathrm{fext}_{\langle 2\rangle}(ab, b)$, $\mathrm{fext}_{\langle 1,1\rangle}(ab, a, a)$, $\mathrm{fext}_{\langle 1,2\rangle}(ab, a, b)$, $\mathrm{fext}_{\langle 2,1\rangle}(ab, b, a)$, $\mathrm{fext}_{\langle 2,2\rangle}(ab, b, b)$, $\mathrm{fext}_{\langle 1,1,1\rangle}(ab, a, a, a)$, ...

  But note that, e.g., $\mathrm{fext}_{\langle 2\rangle}(aa, a)$ is not true even though the tuple $\langle 2\rangle$ extracts the tuple $\langle a\rangle$ from aa, since we can also extract $\langle a\rangle$ from aa with the tuple $\langle 1\rangle$, which is lexicographically smaller than $\langle 2\rangle$.
- $\mathrm{cid}_{\langle 1,1\rangle}(aa, a, a)$, $\mathrm{cid}_{\langle 1,2\rangle}(ab, a, b)$, $\mathrm{cid}_{\langle 2,1\rangle}(ab, b, a)$, $\mathrm{cid}_{\langle 2,2\rangle}(ab, b, b)$, $\mathrm{cid}_{\langle 1,1,1\rangle}(aa, a, a, a)$, $\mathrm{cid}_{\langle 1,1,2\rangle}(ab, a, a, b)$, ...

  But note that, e.g., $\mathrm{cid}_{\langle 1,1\rangle}(ab, a, a)$ is not true since ab is not the first cover of $\langle a, a\rangle$. Also, $\mathrm{cid}_{\langle 2,2\rangle}(aa, a, a)$ is not true since $\langle 2, 2\rangle$ is not the first tuple that extracts $\langle a, a\rangle$ from aa. △

We next show an important property for guarded ASP programs without constants, based on the notions introduced above, namely, that for each possible input and every tuple $\boldsymbol{a}$ of constants that can occur in an atom of the grounding, $\boldsymbol{a}$ has a cover-based identifier.

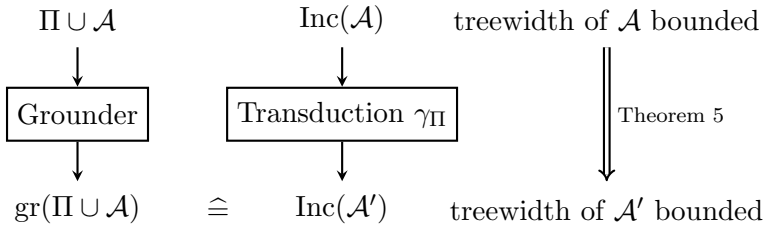


Figure 5: Strategy for proving that grounding a fixed guarded program Π together with an input structure $\mathcal{A}$ preserves bounded treewidth of $\mathcal{A}$. $\mathcal{A}'$ represents the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{A})$.

Lemma 16. *Let Π be a guarded ASP program without constants, $\mathcal{A}$ an input structure of Π and $p(\boldsymbol{a})$ an atom that occurs in $\mathrm{gr}(\Pi \cup \mathcal{A})$. Then the tuple $\boldsymbol{a}$ has a cover-based identifier in $\mathcal{A}$.*

Proof. While defining the notion of cover-based identifier, we have seen that $\boldsymbol{a}$ has such an identifier in $\mathcal{A}$ if a fact covers $\boldsymbol{a}$. Since the atom $p(\boldsymbol{a})$ is part of $\mathrm{gr}(\Pi \cup \mathcal{A})$, it occurs in one of its rules. Let r be a rule in the grounding such that $p(\boldsymbol{a})$ occurs in r, and let r' denote the corresponding non-ground rule. The atom $p(\boldsymbol{a})$ occurs in r as an instantiation of a non-ground atom $p(\boldsymbol{X})$ in r'. Moreover, the positive body of r contains an atom $g(\boldsymbol{b})$ as an instantiation of the guard $g(\boldsymbol{Y})$ of r'. Since $g(\boldsymbol{Y})$ is the guard of r', each variable in $\boldsymbol{X}$ also occurs in $\boldsymbol{Y}$. Hence each element of $\boldsymbol{a}$ is also an element of $\boldsymbol{b}$. Since the fact $g(\boldsymbol{b})$ thus covers $\boldsymbol{a}$, the latter has a cover-based identifier in $\mathcal{A}$. □

We can also prove the related result that for all tuples $\boldsymbol{a}$ and $\boldsymbol{b}$ of constants that can occur in two atoms of the same rule in the grounding, the joint tuple $\boldsymbol{ab}$ has a cover-based identifier.

Lemma 17. *If Π is a guarded ASP program without constants, $\mathcal{A}$ is an input structure of Π and both $p(\boldsymbol{a})$ and $q(\boldsymbol{b})$ are atoms that occur together in a rule of $\mathrm{gr}(\Pi \cup \mathcal{A})$, then the joint tuple $\boldsymbol{ab}$ has a cover-based identifier in $\mathcal{A}$.*

Proof. Since $p(\boldsymbol{a})$ and $q(\boldsymbol{b})$ occur together in a rule r of the grounding, the fact that instantiates the guard in r covers both $\boldsymbol{a}$ and $\boldsymbol{b}$, and therefore it also covers $\boldsymbol{ab}$. □

With the above notions and lemmas defined, we are now ready to prove our main theorem.

Proof of Theorem 14. Our proof strategy is illustrated in Figure 5. In order to formally represent the grounding process and investigate its influence on the treewidth, we use MSO transductions. Our methodology for proving that grounding $\Pi \cup \mathcal{A}$ preserves bounded treewidth of $\mathcal{A}$ works as follows: We study the transformation of $\mathcal{A}$ into the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{A})$. We do this by providing an MSO transduction γ_Π that transforms $\mathrm{Inc}(\mathcal{A})$ into the incidence structure of the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{A})$. By constructing the transduction in such a way that it only depends on Π (and is thus fixed as Π is fixed), we obtain the desired result by virtue of Theorem 5.

We will, for the moment, assume that Π is constant-free; we will later refer to literature showing how to handle the general case. Note that the primal graph is undirected, but we assume that for representing undirected graphs as structures we use symmetric edge relations, so we treat an undirected edge as two directed edges of opposing orientation. Thus, when we speak of the primal graph in this proof, we refer to a directed graph with a symmetric edge relation. The signature of the output structure of our transduction is thus $\{\mathrm{E}, in_1, in_2\}$, where E is unary and the other relations are binary.

We call the domain elements in the output structure that correspond to vertices and edges of the primal graph *vertex elements* and *edge elements*, respectively. In the following, we define γ_Π by the definition scheme $\langle \Delta, \Theta \rangle$, where the tuple Δ is the concatenation of tuples Δ_v and Δ_e, which contain domain formulas that generate the vertex and edge elements, respectively, and Θ contains relation formulas that state which vertex is incident to which edge.[7]

Formulas in Δ_v. These formulas shall produce the vertex elements. For each predicate p of arity k, we first define O_p to be the following set of objects: If a rule r in Π is guarded by an atom $g(X_1, \ldots, X_\ell)$ and contains an atom $p(X_{i_1}, \ldots, X_{i_k})$, then O_p contains $\langle g, i_1, \ldots, i_k \rangle$. With this, we define a formula $\mathrm{occurs}_p(\boldsymbol{x})$, where $\boldsymbol{x}$ is a k-ary tuple of variables, to express that the ground atom $p(\boldsymbol{x})$ occurs in $\mathrm{gr}(\Pi \cup \mathcal{A})$.[8]

$$\mathrm{occurs}_p(\boldsymbol{x}) \;\equiv \bigvee_{\langle g, i_1, \ldots, i_k \rangle \in O_p} \exists y \big(g(y) \wedge \mathrm{extract}_{\langle i_1, \ldots, i_k \rangle}(y, \boldsymbol{x}) \big)$$

We can now define the formula $\delta_{p[\boldsymbol{i}]}(x)$ to be an element of Δ_v, for every predicate p of arity k and each k-ary tuple $\boldsymbol{i}$ of integers between 1 and $\rho_{\max}$.

$$\delta_{p[\boldsymbol{i}]}(x) \;\equiv\; \exists \boldsymbol{y} \big(\mathrm{cid}_{\boldsymbol{i}}(x, \boldsymbol{y}) \wedge \mathrm{occurs}_p(\boldsymbol{y}) \big)$$

This formula is true if and only if x together with $\boldsymbol{i}$ is the cover-based identifier of some tuple $\boldsymbol{a}$ of domain elements and $p(\boldsymbol{a})$ occurs in $\mathrm{gr}(\Pi \cup \mathcal{A})$; the resulting copy of x in the output structure then corresponds to the atom $p(\boldsymbol{a})$.

For each atom $p(\boldsymbol{a})$ in the grounding, we thus produce a vertex element, since $\boldsymbol{a}$ has a cover-based identifier by Lemma 16. Moreover, different atoms produce different vertex elements: If they have different arguments, they have different cover-based identifiers, and otherwise they differ in their predicate symbol. In both cases, they clearly produce different copies. Finally, every vertex element that we produce corresponds to an atom in the grounding by our construction of the occurs_p formulas. This proves that there is a bijection between the atoms in the grounding and the vertex elements.

Formulas in Δ_e. These formulas shall produce the edge elements. Before we define them, we introduce an auxiliary formula. For all predicates p and q of arity k and ℓ, respectively,

[7] As mentioned in Section 2.3, the formulas in Δ and Θ are denoted by symbols with subscripts, which rely on a set I. For our purpose, i.e., for defining γ_Π, we choose as I the set containing (a) an element $p[\boldsymbol{i}]$ for every predicate p of arity k and each k-ary tuple $\boldsymbol{i}$ of integers between 1 and $\rho_{\max}$, and (b) an element $p[\boldsymbol{i}]q[\boldsymbol{j}]$ for all predicates p and q, and all tuples $\boldsymbol{i}$ and $\boldsymbol{j}$ of integers between 1 and $\rho_{\max}$, such that the arities of $\boldsymbol{i}$ and $\boldsymbol{j}$ are the same as those of p and q, respectively. All elements of Δ and Θ that we do not explicitly mention are defined as $\bot$.

[8] We slightly abuse notation by sometimes using tuples where lists would be required.

we define $T_{p,q}$ to be the following set of objects: If there is a rule r in Π guarded by an atom $g(X_1, \ldots, X_m)$ such that r contains two atoms $p(X_{i_1}, \ldots, X_{i_k})$ and $q(X_{j_1}, \ldots, X_{j_\ell})$, then $T_{p,q}$ contains $\langle g, \boldsymbol{i}, \boldsymbol{j}\rangle$, where $\boldsymbol{i} = \langle i_1, \ldots, i_k\rangle$ and $\boldsymbol{j} = \langle j_1, \ldots, j_\ell\rangle$. With this, we now define a formula $\text{together}_{p,q}(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{x}$ and $\boldsymbol{y}$ are tuples of variables with arity k and ℓ, respectively. This formula expresses that the two ground atoms $p(\boldsymbol{x})$ and $q(\boldsymbol{y})$ occur together in some rule of $\text{gr}(\Pi \cup \mathcal{A})$.

$$\text{together}_{p,q}(\boldsymbol{x}, \boldsymbol{y}) \equiv \bigvee_{\langle g, \boldsymbol{i}, \boldsymbol{j}\rangle \in T_{p,q}} \exists z\big(g(z) \wedge \text{extract}_{\boldsymbol{i}}(z, \boldsymbol{x}) \wedge \text{extract}_{\boldsymbol{j}}(z, \boldsymbol{y})\big)$$

With this auxiliary formula in hand, we define the following formula $\delta_{p[\boldsymbol{i}]q[\boldsymbol{j}]}(x)$ to be an element of Δ_e, for all predicates p and q, and all tuples $\boldsymbol{i}$ and $\boldsymbol{j}$ of integers between 1 and $\rho_{\max}$, such that the arities of $\boldsymbol{i}$ and $\boldsymbol{j}$ are the same as those of p and q, respectively.

$$\delta_{p[\boldsymbol{i}]q[\boldsymbol{j}]}(x) \equiv \begin{cases} \bot & \text{if } p[\boldsymbol{i}] = q[\boldsymbol{j}] \\ \exists \boldsymbol{y} \exists \boldsymbol{z}\big(\text{cid}_{\boldsymbol{ij}}(x, \boldsymbol{y}, \boldsymbol{z}) \wedge \text{together}_{p,q}(\boldsymbol{y}, \boldsymbol{z})\big) & \text{otherwise} \end{cases}$$

This formula is true if and only if there are tuples $\boldsymbol{a}$ and $\boldsymbol{b}$ of the same arity as p and q, respectively, such that (1) x together with $\boldsymbol{ij}$ is the cover-based identifier of $\boldsymbol{ab}$, and (2) the atoms $p(\boldsymbol{a})$ and $q(\boldsymbol{b})$ are different and occur together in some rule of $\text{gr}(\Pi \cup \mathcal{A})$. The resulting copy of x in the output structure then corresponds to the edge from $p(\boldsymbol{a})$ to $q(\boldsymbol{b})$ in the primal graph. Due to symmetry, we can see that then also an edge in the other direction will be created. Since both atoms are different if the formula is true, we do not introduce self-loops.

For each pair of different atoms $p(\boldsymbol{a})$ and $q(\boldsymbol{b})$ that jointly occur in a rule of the grounding, we thus correctly produce two edge elements, since $\boldsymbol{ab}$ has a cover-based identifier by Lemma 17. Moreover, different such pairs of atoms produce different edge elements, and every edge element that we produce corresponds to a joint occurrence of two different atoms in a rule of the grounding by our construction of the $\text{together}_{p,q}$ formulas.

Formulas in Θ. These formulas shall ensure that each edge element is incident to the two appropriate vertex elements. First, let p and q be predicates occurring in Π, and let $\boldsymbol{i}$ and $\boldsymbol{j}$ be tuples of integers between 1 and $\rho_{\max}$ such that $\boldsymbol{i}$ and $\boldsymbol{j}$ have the same arity as p and q, respectively. We define a formula $\text{eq}_{p[\boldsymbol{i}],q[\boldsymbol{j}]}(x, y)$ to express that the atoms $p(\boldsymbol{a})$ and $q(\boldsymbol{b})$ are equal, where $\boldsymbol{a}$ is the tuple extracted from x by $\boldsymbol{i}$, and $\boldsymbol{b}$ is the tuple extracted from y by $\boldsymbol{j}$.

$$\text{eq}_{p[\boldsymbol{i}],q[\boldsymbol{j}]}(x, y) \equiv \begin{cases} \exists \boldsymbol{z}\big(\text{extract}_{\boldsymbol{i}}(x, \boldsymbol{z}) \wedge \text{extract}_{\boldsymbol{j}}(y, \boldsymbol{z})\big) & \text{if } p = q \\ \bot & \text{otherwise} \end{cases}$$

Let p, q and q' be predicates occurring in Π, and let $\boldsymbol{i}$, $\boldsymbol{j}$ and $\boldsymbol{j}'$ be tuples of integers between 1 and $\rho_{\max}$ with the same arity as p, q and q', respectively. We define the following formulas to be elements of Θ[9]:

$$\theta_{in_1, p[\boldsymbol{i}], q[\boldsymbol{j}]q'[\boldsymbol{j}']}(x, y) \equiv \text{eq}_{p[\boldsymbol{i}],q[\boldsymbol{j}]}(x, y)$$
$$\theta_{in_2, p[\boldsymbol{i}], q[\boldsymbol{j}]q'[\boldsymbol{j}']}(x, y) \equiv \text{eq}_{p[\boldsymbol{i}],q'[\boldsymbol{j}']}(x, y)$$

[9] If we do not explicitly mention relation formulas like the remaining relation formulas for defining in_1 and in_2 (those having subscripts of different forms than the shown formulas), then these are defined as $\bot$.

We only explain the first of these formulas, as the other case is symmetric; instead of outgoing edges (due to in_1 in the subscript) it concerns incoming edges (due to in_2 in the subscript).

The formula $\theta_{in_1,p[\boldsymbol{i}],q[\boldsymbol{j}]q'[\boldsymbol{j}']}(x,y)$ is true if and only if the atom $p(\boldsymbol{a})$ is equal to $q(\boldsymbol{b})$, where $\boldsymbol{a}$ is the tuple extracted from x by $\boldsymbol{i}$, and $\boldsymbol{b}$ is the tuple extracted from y by $\boldsymbol{j}$. If this formula is true, it makes the edge represented by the respective copy of y an *outgoing* edge of $p(\boldsymbol{a})$ because of the subscript in_1.

We first show that, whenever this formula causes an edge element to be incident to a vertex element, the corresponding edge in the primal graph is indeed an outgoing edge of the appropriate vertex. Suppose there are predicates p, q and p', tuples $\boldsymbol{i}$, $\boldsymbol{j}$ and $\boldsymbol{j}'$, as well as domain elements x and y such that (1) $\delta_{p[\boldsymbol{i}]}(x)$ is true, (2) $\delta_{q[\boldsymbol{j}]q'[\boldsymbol{j}']}(y)$ is true, and (3) $\theta_{in_1,p[\boldsymbol{i}],q[\boldsymbol{j}]q'[\boldsymbol{j}']}(x,y)$ is true. As observed in our definition of Δ_v, (1) means that x together with $\boldsymbol{i}$ is the cover-based identifier of some tuple $\boldsymbol{a}$, and the grounding contains an atom $p(\boldsymbol{a})$. From (2) we get that there are tuples $\boldsymbol{b}$ and $\boldsymbol{b}'$ such that y together with $\boldsymbol{j}\boldsymbol{j}'$ is the cover-based identifier of $\boldsymbol{b}\boldsymbol{b}'$. We have also seen that by (2) there is a rule in the grounding that contains both $q(\boldsymbol{b})$ and $q'(\boldsymbol{b}')$, hence there is an edge from $q(\boldsymbol{b})$ to $q'(\boldsymbol{b}')$ in the primal graph. By (3) and the definition of cover-based identifiers (in particular those of $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{b}'$), we know that $p(\boldsymbol{a})$ is equal to $q(\boldsymbol{b})$. Hence the edge in the primal graph from $q(\boldsymbol{b})$ to $q'(\boldsymbol{b}')$ is indeed an outgoing edge of $p(\boldsymbol{a})$.

Finally we prove the other direction: Whenever an edge in the primal graph is an outgoing edge of a vertex, a formula in Θ defining the relation in_1 causes the corresponding edge element to be an outgoing edge of the appropriate vertex element. Suppose that the primal graph contains an edge from atom $p(\boldsymbol{a})$ to atom $q(\boldsymbol{b})$. Then these atoms occur together in a rule of the grounding, so there is a non-ground rule r guarded by an atom $g(X_1,\dots,X_m)$ such that r contains both $p(X_{i_1},\dots,X_{i_k})$ and $q(X_{j_1},\dots,X_{j_\ell})$, and $\mathcal{A}$ contains a fact $g(c_1,\dots,c_m)$ such that $\boldsymbol{a}=\langle c_{i_1},\dots,c_{i_k}\rangle$ and $\boldsymbol{b}=\langle c_{j_1},\dots,c_{j_\ell}\rangle$. Moreover, we have seen in our definition of Δ_v and Δ_e that then our transduction produces a vertex element v for $p(\boldsymbol{a})$ and an edge element e for the edge from $p(\boldsymbol{a})$ to $q(\boldsymbol{b})$. Now let x and y be domain elements of $\mathcal{A}$, and let $\boldsymbol{i}$, $\boldsymbol{j}$ and $\boldsymbol{j}'$ be tuples of integers, such that x together with $\boldsymbol{i}$ is the cover-based identifier of $\boldsymbol{a}$, and y together with $\boldsymbol{j}\boldsymbol{j}'$ is the cover-based identifier of $\boldsymbol{a}\boldsymbol{b}$. By definition of cover-based identifiers, $\boldsymbol{i}$ extracts $\boldsymbol{a}$ from x; moreover, $\boldsymbol{j}$ and $\boldsymbol{j}'$ extract $\boldsymbol{a}$ and $\boldsymbol{b}$ from y, respectively. Since $\boldsymbol{a}$ can be extracted from x by $\boldsymbol{i}$, as well as from y by $\boldsymbol{j}$, the formula $\mathrm{eq}_{p[\boldsymbol{i}],p[\boldsymbol{j}]}(x,y)$ is clearly true. Hence the formula $\theta_{in_1,p[\boldsymbol{i}],p[\boldsymbol{j}]q[\boldsymbol{j}']}(x,y)$ is true, which correctly makes the edge element e an outgoing edge of the vertex element v.

We are still missing the relation formulas for defining the remaining unary relation E, which identifies the edge elements. This is easy: We can just set $\theta_{\mathrm{E},p[\boldsymbol{i}]}$ to $\bot$ (for all $p[\boldsymbol{i}]$ as before) and $\theta_{\mathrm{E},p[\boldsymbol{i}]q[\boldsymbol{j}]}$ to $\top$.

This completes the construction of the MSO transduction γ_Π. Let $\mathcal{A}$ be an input structure for Π of bounded treewidth. We have argued that $\gamma_\Pi(\mathrm{Inc}(\mathcal{A}))$ yields the incidence structure of the primal graph of $\mathrm{gr}(\Pi\cup\mathcal{A})$ as desired. By Theorem 5 and the fact that $\mathcal{A}$, by assumption, has bounded treewidth, this proves our claim for constant-free programs.

We can also generalize this proof to programs with constants (Bliem, 2017). □

We now illustrate this transduction for an example program.

Example 18. Let Π be the following guarded program for solving the 2-Colorability problem on directed graphs, where the input graph is given via the binary edge predicate $\mathtt{e}$, and the colors are $\mathtt{r}$ and $\mathtt{g}$:

$$\mathtt{r}(\mathtt{X}) \vee \mathtt{g}(\mathtt{X}) \leftarrow \mathtt{e}(\mathtt{X},\mathtt{Y}).$$
$$\mathtt{r}(\mathtt{Y}) \vee \mathtt{g}(\mathtt{Y}) \leftarrow \mathtt{e}(\mathtt{X},\mathtt{Y}).$$
$$\leftarrow \mathtt{e}(\mathtt{X},\mathtt{Y}),\ \mathtt{r}(\mathtt{X}),\ \mathtt{r}(\mathtt{Y}).$$
$$\leftarrow \mathtt{e}(\mathtt{X},\mathtt{Y}),\ \mathtt{g}(\mathtt{X}),\ \mathtt{g}(\mathtt{Y}).$$

Next, let $\mathcal{G}$ be the input structure for Π that represents a graph consisting of vertices a and b, with an edge from b to a. Moreover, we assume that the order on the domain elements is such that $a < b$. That is, $\mathrm{dom}(\mathcal{G}) = \{a, b\}$, $succ^{\mathcal{G}} = \{\langle a, b\rangle\}$ and $\mathtt{e}^{\mathcal{G}} = \{\langle b, a\rangle\}$. We denote the domain element of $\mathrm{Inc}(\mathcal{G})$ for the fact $\langle b, a\rangle$ in $\mathtt{e}^{\mathcal{G}}$ and for $\langle a, b\rangle$ in $succ^{\mathcal{G}}$ by ba and ab, respectively, and we assume the ordering $a < b < ab < ba$. We show how γ_Π transforms $\mathrm{Inc}(\mathcal{G})$ into the incidence structure of the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{G})$.

For the vertex elements, first observe that all of the formulas $\mathrm{occurs}_{\mathtt{e}}(b, a)$, $\mathrm{occurs}_{\mathtt{r}}(a)$, $\mathrm{occurs}_{\mathtt{r}}(b)$, $\mathrm{occurs}_{\mathtt{g}}(a)$ and $\mathrm{occurs}_{\mathtt{g}}(b)$ are true. For instance, to see that $\mathrm{occurs}_{\mathtt{e}}(b, a)$ is true, observe that $O_{\mathtt{e}} = \{\langle \mathtt{e}, 1, 2\rangle\}$ (under the assumption that the first variable of each rule is $\mathtt{X}$ and the second is $\mathtt{Y}$). The definition of $\mathrm{occurs}_{\mathtt{e}}(b, a)$ thus boils down to

$$\mathrm{occurs}_{\mathtt{e}}(b, a) \ \equiv\ \exists y\big(\mathtt{e}(y) \wedge \mathrm{extract}_{\langle 1,2\rangle}(y, b, a)\big).$$

Clearly $\mathtt{e}(ba) \wedge \mathrm{extract}_{\langle 1,2\rangle}(ba, b, a)$ is true, so $\mathrm{occurs}_{\mathtt{e}}(b, a)$ is true.

Next we show that we correctly construct vertex elements corresponding to the atoms $\mathtt{e}(b, a)$, $\mathtt{r}(a)$ and $\mathtt{r}(b)$ in the grounding. Since the cover-based identifier of $\langle b, a\rangle$ is ab in combination with $\langle 2, 1\rangle$, the formula $\mathrm{cid}_{\langle 2,1\rangle}(ab, b, a)$ is true. We can conclude that $\delta_{\mathtt{e}[2,1]}(ab)$ is true by looking at its definition

$$\delta_{\mathtt{e}[2,1]}(ab) \ \equiv\ \exists y_1 \exists y_2\big(\,\mathrm{cid}_{\langle 2,1\rangle}(ab, y_1, y_2) \wedge \mathrm{occurs}_{\mathtt{e}}(y_1, y_2)\big)$$

and observing that $\mathrm{cid}_{\langle 2,1\rangle}(ab, b, a) \wedge \mathrm{occurs}_{\mathtt{e}}(b, a)$ is true. We thus produce the vertex element for $\mathtt{e}(b, a)$.

Similarly, $O_{\mathtt{r}} = \{\langle \mathtt{e}, 1\rangle, \langle \mathtt{e}, 2\rangle\}$, and the cover-based identifier of $\langle a\rangle$ and $\langle b\rangle$ is ab together with $\langle 1\rangle$ and $\langle 2\rangle$, respectively. This makes $\delta_{\mathtt{r}[1]}(ab)$ and $\delta_{\mathtt{r}[2]}(ab)$ true and produces the vertex elements for $\mathtt{r}(a)$ and $\mathtt{r}(b)$, respectively.

We illustrate the edge elements just by the construction for the edge from atom $\mathtt{r}(a)$ to atom $\mathtt{e}(b, a)$. Recall that $\mathrm{together}_{\mathtt{r},\mathtt{e}}(\langle a\rangle, \langle b, a\rangle)$ is defined as

$$\mathrm{together}_{\mathtt{r},\mathtt{e}}(\langle a\rangle, \langle b, a\rangle) \ \equiv \bigvee_{\langle g, \boldsymbol{i}, \boldsymbol{j}\rangle \in T_{\mathtt{r},\mathtt{e}}} \exists z\big(g(z) \wedge \mathrm{extract}_{\boldsymbol{i}}(z, a) \wedge \mathrm{extract}_{\boldsymbol{j}}(z, b, a)\big).$$

Observe that this formula is true because $T_{\mathtt{r},\mathtt{e}}$ contains the tuple $\langle \mathtt{e}, \langle 2\rangle, \langle 1, 2\rangle\rangle$ and clearly $\mathtt{e}(ba) \wedge \mathrm{extract}_{\langle 2\rangle}(ba, a) \wedge \mathrm{extract}_{\langle 1,2\rangle}(ba, b, a)$ is true. Moreover, observe that ab together with $\langle 1, 2, 1\rangle$ is the cover-based identifier of $\langle a, b, a\rangle$. We can see that $\delta_{\mathtt{r}[1]\mathtt{e}[2,1]}(ab)$ is true by looking at its definition

$$\delta_{\mathtt{r}[1]\mathtt{e}[2,1]}(ab) \ \equiv\ \exists y_1 \exists z_1 \exists z_2\big(\,\mathrm{cid}_{\langle 1,2,1\rangle}(ab, y_1, z_1, z_2) \wedge \mathrm{together}_{\mathtt{r},\mathtt{e}}(\langle y_1\rangle, \langle z_1, z_2\rangle)\big)$$

and observing that $\mathrm{cid}_{\langle 1,2,1\rangle}(ab, a, b, a) \wedge \mathrm{together}_{\mathtt{r},\mathtt{e}}(\langle a\rangle, \langle b, a\rangle)$ is true. Thus we produce the desired edge element.

Finally we show that our transduction indeed makes the edge that should go from $\mathtt{r}(a)$ to $\mathtt{e}(b,a)$ an outgoing edge of $\mathtt{r}(a)$. Recall that the vertex element for $\mathtt{r}(a)$ exists due to $\delta_{\mathtt{r}[1]}(ab)$ being true, and the edge element exists due to $\delta_{\mathtt{r}[1]\mathtt{e}[2,1]}(ab)$ being true. We can see that $\mathrm{eq}_{\mathtt{r}[1],\mathtt{r}[1]}(ab,ab)$ is true be looking at its its definition

$$\mathrm{eq}_{\mathtt{r}[1],\mathtt{r}[1]}(ab,ab) \;\equiv\; \exists z\big(\,\mathrm{extract}_{\langle 1\rangle}(ab,z) \wedge \mathrm{extract}_{\langle 1\rangle}(ab,z)\big)$$

and observing that $\mathrm{extract}_{\langle 1\rangle}(ab,a) \wedge \mathrm{extract}_{\langle 1\rangle}(ab,a)$ is true. Now we can immediately conclude that $\theta_{in_1,\,\mathtt{r}[1],\,\mathtt{r}[1]\mathtt{e}[2,1]}(ab,ab)$ is true by considering its definition

$$\theta_{in_1,\,\mathtt{r}[1],\,\mathtt{r}[1]\mathtt{e}[2,1]}(ab,ab) \;\equiv\; \mathrm{eq}_{\mathtt{r}[1],\mathtt{r}[1]}(ab,ab).$$

This makes the vertex incident to the edge element as desired. △

Alternatively, we can prove that guarded encodings preserve bounded treewidth in a more direct way by modifying a tree decomposition of the input so that the result is a tree decomposition of the grounding. This gives us a simpler, more elementary proof and it also allows us to derive an explicit bound on the treewidth of the grounding. Nevertheless, the preceding proof based on MSO transductions may be of interest because it facilitates the understanding of the MSO transduction in Section 4.3, which is more complex.

Theorem 19. *If Π is a guarded ASP program containing c constants and k predicates of arity at most ℓ, and $\mathcal{A}$ is an input structure of Π having treewidth w, then the treewidth of the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{A})$ is at most $k \cdot (w + c + 1)^{\ell} - 1$.*

Proof. Let $\mathcal{T}$ be a tree decomposition of $\mathcal{A}$ having width w, and let C denote the constants in Π. We construct a tree decomposition $\mathcal{T}'$ having width $k \cdot (w+c+1)^{\ell} - 1$ of a supergraph of the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{A})$. Since the treewidth of a subgraph is at most the treewidth of the whole graph, the statement follows.

We define the tree in $\mathcal{T}'$ to be isomorphic to the tree in $\mathcal{T}$. Let N be a node in $\mathcal{T}$ and B be its bag. We define the bag B' of the corresponding node N' in $\mathcal{T}'$ to consist of all atoms $p(\boldsymbol{x})$ such that p is a predicate occurring in Π and $\boldsymbol{x}$ is a tuple of elements of $B \cup C$. The size of B' is then at most $k \cdot (w + c + 1)^{\ell}$. It remains to show that $\mathcal{T}'$ is indeed a tree decomposition of a supergraph of the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{A})$.

For every atom $p(\boldsymbol{x})$ in a rule r of the grounding, we know from guardedness that there is a ground atom $g(\boldsymbol{y})$ in the positive body of r such that g is extensional and every element of $\boldsymbol{x}$ that is not a constant is also an element of $\boldsymbol{y}$. Since g is extensional, there is a node in $\mathcal{T}$ whose bag contains all elements of $\boldsymbol{y}$. By our construction, the bag of the corresponding node in $\mathcal{T}'$ contains $p(\boldsymbol{x})$.

If two atoms $p(\boldsymbol{x})$ and $q(\boldsymbol{y})$ occur together in a rule r of the grounding, then from guardedness we infer that r also contains an atom $g(\boldsymbol{z})$ in the positive body of r such that g is extensional and every element of $\boldsymbol{x}$ or $\boldsymbol{y}$ that is not a constant is also an element of $\boldsymbol{z}$. As before, it follows that the bag of a node in $\mathcal{T}$ contains all elements of $\boldsymbol{x}$ and $\boldsymbol{y}$ that are not constants, and the bag of the corresponding node in $\mathcal{T}'$ contains both $p(\boldsymbol{x})$ and $q(\boldsymbol{y})$.

If the bags of two nodes N', M' of $\mathcal{T}'$ both contain an atom $p(\boldsymbol{x})$, then the bags of the corresponding nodes N, M in $\mathcal{T}$ contain all elements of $\boldsymbol{x}$ that are not constants. By the connectedness condition, every bag of each node between N and M in $\mathcal{T}$ contains all

elements of $\boldsymbol{x}$ that are not constants. Hence, by our construction, the bags of all nodes between N' and M' in $\mathcal{T}'$ contain $p(\boldsymbol{x})$. This proves that $\mathcal{T}'$ is a tree decomposition of a supergraph of the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{A})$, and its width is at most $k \cdot (w + c + 1)^{\ell} - 1$. □

Since the guarded program Π and thus c, k and ℓ are fixed, this shows that the treewidth of the primal graph of $\mathrm{gr}(\pi \cup \mathcal{A})$ is polynomial in the treewidth of the input $\mathcal{A}$.

4.3 Connection-Guarded Answer Set Programs

In this section, we define the class of connection-guarded ASP programs and show that they lead to groundings whose treewidth depends only on the treewidth and the degree of the input structure. We first require the following concept.

Definition 20. The *join structure* of a set S of atoms is the following relational structure $\mathcal{J}$ over the signature consisting of the predicate symbols occurring in S, with the same respective arities as in S. The domain of $\mathcal{J}$ consists of the variables and constants occurring in S and for each predicate symbol p it holds that $p^{\mathcal{J}} = \{\boldsymbol{t} \mid p(\boldsymbol{t}) \in S\}$. The *join graph* of S is the Gaifman graph of the join structure of S.

With this notion in hand, we can define our ASP class of interest.

Definition 21. Let Π be an ASP program. A *connection-guard* of a rule r in Π is a set G of extensional atoms occurring in $B^+(r)$ such that all variables that occur in r also occur in G and the join graph of G is connected. We call Π *connection-guarded* if every rule has a connection-guard. We designate an arbitrary connection-guard of r as *the connection-guard* of r.

As mentioned above, the intention of connection-guarded programs is to guarantee that the treewidth of the grounding remains bounded, provided that the treewidth and degree of the input instance is also bounded. The following theorem is the main result of this section and states this formally.

Theorem 22. *Let Π be a fixed connection-guarded program and let $\mathcal{A}$ be an input structure of Π. If $\mathcal{A}$ has bounded treewidth and degree, then the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{A})$ has bounded treewidth.*

Again, we need to introduce several intermediate notions and lemmas in order to establish correctness of the statement above in a formal proof. To this end, recall the notions defined in Section 4.2, which we will reuse in the present section. Furthermore, we now introduce several additional formulas that enable us to identify certain tuples $\langle a_1, \ldots, a_k \rangle$ by a "source" element s together with a tuple of objects $\langle \pi_1, \ldots, \pi_k \rangle$, where each π_j determines a path from s to a_j in $\mathcal{A}$. Note that this is the idea underlying the class of connection-guarded ASP programs: only if some domain elements are reachable, via a constant number of steps, from a fact in $\mathcal{A}$ can they give rise to a new ground atom in $\mathrm{gr}(\Pi \cup \mathcal{A})$.

First we define a formula to express that x and y are neighbors in the sense that they are adjacent to each other in the Gaifman graph of the base structure (cf. Section 2.2).

$$\mathrm{neigh}(x, y) \;\equiv\; x \neq y \wedge \exists z \big(\mathit{in}(z, x) \wedge \mathit{in}(z, y) \big)$$

The order of the domain elements of a structure induces an order on the neighborhood of each domain element. For each positive integer i, the following formula expresses that y is the i-th neighbor of x according to this order.

$$\mathrm{neigh}_i(x,y) \;\equiv\; \mathrm{neigh}(x,y) \wedge \forall z\big(z < y \wedge \mathrm{neigh}(x,z) \rightarrow \bigvee_{1\leq j<i} \mathrm{neigh}_j(x,z)\big)$$

A *relative path* of length k is a k-tuple of positive integers. It is called a *relative d-path* if each integer in the tuple is less than or equal to d. Moreover, we say that a relative d-path is a *relative (ℓ,d)-path* if its length is at most ℓ. A *path* of length k between two domain elements x and y of $\mathcal{A}$ is a sequence of domain elements $a_0, a_1, \ldots, a_k$ such that $a_0 = x$, $a_k = y$, and a_j is a neighbor of a_{j-1}, for $1 \leq j \leq k$. Each path can be uniquely identified by a relative path π together with a starting point $s \in \mathrm{dom}(\mathcal{A})$ in the obvious way by starting from the first element of the path, and we also write s^π to denote the end point of this path. For a tuple $\boldsymbol{\pi} = \langle \pi_1, \ldots, \pi_k \rangle$ of relative paths, we write $s^{\boldsymbol{\pi}}$ to denote $\langle s^{\pi_1}, \ldots, s^{\pi_k} \rangle$. We say that a path $\boldsymbol{p}$ is a *d-path* if its corresponding relative path is a relative d-path, and we call $\boldsymbol{p}$ an *(ℓ,d)-path* if it is a d-path of length at most ℓ.

For every integer k and each relative path $\pi = \langle i_1, \ldots, i_k \rangle$, we define the formula $\mathrm{reach}_\pi(x,y)$ to express that x^π is defined and equal to y. We write ε to denote the relative path of length 0.

$$\begin{aligned} \mathrm{reach}_\varepsilon(x,y) \;&\equiv\; x = y \\ \mathrm{reach}_{\langle i_1,\ldots,i_k\rangle}(x,y) \;&\equiv\; \exists z\big(\,\mathrm{reach}_{\langle i_1,\ldots,i_{k-1}\rangle}(x,z) \wedge \mathrm{neigh}_{i_k}(z,y)\big) \quad \text{for } k \geqslant 1 \end{aligned}$$

Since two domain elements may be connected via more than one path, we next describe how we can designate a single representative among them. We define the strict total order $\prec_d$ over relative d-paths such that $\pi \prec_d \psi$ if π is shorter than ψ or they have the same length but π is lexicographically smaller than ψ. If there is a d-path from a to b in $\mathcal{A}$, we define the *first relative d-path* from a to b as the smallest relative d-path π such that $a^\pi = b$, where "smallest" refers to $\prec_d$. For each nonnegative integer d and every relative d-path π, we now define the formula $\mathrm{frp}^d_\pi(x,y)$ to represent that π is the first relative d-path from x to y. (Note that the intended meaning of the abbreviation "frp" is "first relative path".)

$$\mathrm{frp}^d_\pi(x,y) \;\equiv\; \mathrm{reach}_\pi(x,y) \wedge \bigwedge_{\psi \prec_d \pi} \neg\, \mathrm{reach}_\psi(x,y)$$

Let ℓ, d and k be arbitrary nonnegative integers, and $\boldsymbol{a}$ be a tuple of domain elements such that there is a domain element from which there is an (ℓ,d)-path to each element of $\boldsymbol{a}$. We want to be able to identify $\boldsymbol{a}$ by a combination of a domain element s and a tuple $\boldsymbol{\pi}$ of relative (ℓ,d)-paths such that $s^{\boldsymbol{\pi}} = \boldsymbol{a}$. Of course, there may be multiple choices for s and $\boldsymbol{\pi}$ that identify $\boldsymbol{a}$ in such a way, so we want to single out a unique representative. To this end, we define the *(ℓ,d)-path-based identifier* of $\boldsymbol{a}$ as the unique combination of a domain element s and a tuple of relative (ℓ,d)-paths $\boldsymbol{\pi}$ that satisfies the following properties:

1. For every i, the i-th element of $\boldsymbol{\pi}$ is the first relative d-path from s to the i-th element of $\boldsymbol{a}$.

2. There is no domain element t smaller than s for which there is a tuple $\boldsymbol{\psi}$ of relative (ℓ, d)-paths such that $t^{\boldsymbol{\psi}} = \boldsymbol{a}$.

We now define the formula $\mathrm{pid}^{\ell,d}_{\langle \pi_1,\ldots,\pi_k\rangle}(x, y_1, \ldots, y_k)$ to express that x together with the tuple $\langle \pi_1, \ldots, \pi_k\rangle$ is the (ℓ, d)-path-based identifier of $\langle y_1, \ldots, y_k\rangle$. For this, let $P^k_{\ell,d}$ denote the set of all k-tuples of relative (ℓ, d)-paths. Note that this set is finite.

$$\mathrm{pid}^{\ell,d}_{\langle \pi_1,\ldots,\pi_k\rangle}(x, y_1, \ldots, y_k) \equiv \bigwedge_{1\leq i\leq k} \mathrm{frp}^d_{\pi_i}(x, y_i) \wedge \neg\exists z \left(z < x \wedge \bigvee_{\langle \psi_1,\ldots,\psi_k\rangle \in P^k_{\ell,d}} \bigwedge_{1\leq i\leq k} \mathrm{frp}^d_{\psi_i}(z, y_i)\right)$$

Note that for the empty tuple ε, the formula $\mathrm{pid}^{\ell,d}_{\varepsilon}(x)$ becomes $\neg\exists z\,(z < x)$, which is true if and only if x is the smallest domain element.

Any tuple $\boldsymbol{a}$ of domain elements has an (ℓ, d)-path-based identifier whenever there is a domain element s and a tuple of relative (ℓ, d)-paths $\boldsymbol{\pi}$ such that $\boldsymbol{a} = s^{\boldsymbol{\pi}}$: Then for each $a \in \boldsymbol{a}$, there is an (ℓ, d)-path from s to a, so there is also a first relative (ℓ, d)-path from s to a. Moreover, for every domain element s and each tuple $\boldsymbol{\pi}$ of relative (ℓ, d)-paths, the combination of s and $\boldsymbol{\pi}$ is the (ℓ, d)-path-based identifier of at most one tuple of domain elements. Hence there is a bijection between the set of all tuples that have an (ℓ, d)-path-based identifier and the set of all (ℓ, d)-path-based identifiers.

Example 23. Let $\mathcal{A}$ be a structure over a signature consisting of the binary relation symbols R and *succ* such that $\mathrm{dom}(\mathcal{A}) = \{a, b, c, d\}$, $R^{\mathcal{A}} = \{\langle a, b\rangle, \langle a, d\rangle, \langle b, c\rangle, \langle d, c\rangle\}$ and $succ^{\mathcal{A}} = \{\langle a, b\rangle, \langle b, c\rangle, \langle c, d\rangle\}$. We assume that $a < b < c < d$ holds for the ordering of the domain elements of $\mathrm{Inc}(\mathcal{A})$, and that the domain elements corresponding to facts are all greater than d – their exact order is irrelevant for this example. The following formulas are true under $\mathrm{Inc}(\mathcal{A})$:

- $\mathrm{neigh}(a, b)$, $\mathrm{neigh}(a, d)$, $\mathrm{neigh}(b, c)$, $\mathrm{neigh}(d, c)$.
  $\mathrm{neigh}(b, a)$, $\mathrm{neigh}(d, a)$, $\mathrm{neigh}(c, b)$, $\mathrm{neigh}(c, d)$.
- $\mathrm{neigh}_1(a, b)$, $\mathrm{neigh}_2(a, d)$, $\mathrm{neigh}_1(b, a)$, $\mathrm{neigh}_2(b, c)$, $\mathrm{neigh}_1(c, b)$, $\mathrm{neigh}_2(c, d)$.
  $\mathrm{neigh}_1(d, a)$, $\mathrm{neigh}_2(d, c)$
- $\mathrm{reach}_{\varepsilon}(a, a)$, $\mathrm{reach}_{\langle 1\rangle}(a, b)$, $\mathrm{reach}_{\langle 2\rangle}(a, d)$, $\mathrm{reach}_{\langle 1,1\rangle}(a, a)$, $\mathrm{reach}_{\langle 1,2\rangle}(a, c)$, $\mathrm{reach}_{\langle 2,1\rangle}(a, a)$, $\mathrm{reach}_{\langle 2,2\rangle}(a, c)$, $\mathrm{reach}_{\langle 1,1,1\rangle}(a, b)$, …
  $\mathrm{reach}_{\varepsilon}(b, b)$, $\mathrm{reach}_{\langle 1\rangle}(b, a)$, $\mathrm{reach}_{\langle 2\rangle}(b, c)$, $\mathrm{reach}_{\langle 1,1\rangle}(b, b)$, …
  $\mathrm{reach}_{\varepsilon}(c, c)$, $\mathrm{reach}_{\langle 1\rangle}(c, b)$, $\mathrm{reach}_{\langle 2\rangle}(c, d)$, $\mathrm{reach}_{\langle 1,1\rangle}(c, a)$, …
  $\mathrm{reach}_{\varepsilon}(d, d)$, $\mathrm{reach}_{\langle 1\rangle}(d, a)$, $\mathrm{reach}_{\langle 2\rangle}(d, c)$, $\mathrm{reach}_{\langle 1,1\rangle}(d, b)$, …
- $\mathrm{frp}^1_{\varepsilon}(a, a)$, $\mathrm{frp}^1_{\langle 1\rangle}(a, b)$.
  But note that, e.g., $\mathrm{frp}^1_{\langle 1,1\rangle}(a, a)$ is not true even though $\langle 1, 1\rangle$ is a relative 1-path and it leads to a when starting from a, since so does ε, which is lexicographically smaller than $\langle 1, 1\rangle$. Moreover, neither $\mathrm{frp}^1_{\pi}(a, c)$ nor $\mathrm{frp}^1_{\pi}(a, d)$ are true for any relative 1-path π, since going from a to c or d requires visiting the "second neighbor" of at least one element.

$\mathrm{frp}^1_{\langle 1\rangle}(b,a)$, $\mathrm{frp}^1_{\varepsilon}(b,b)$.

$\mathrm{frp}^1_{\langle 1,1\rangle}(c,a)$, $\mathrm{frp}^1_{\langle 1\rangle}(c,b)$, $\mathrm{frp}^1_{\varepsilon}(c,c)$.

$\mathrm{frp}^1_{\langle 1\rangle}(d,a)$, $\mathrm{frp}^1_{\varepsilon}(d,d)$.

$\mathrm{frp}^2_{\varepsilon}(a,a)$, $\mathrm{frp}^2_{\langle 1\rangle}(a,b)$, $\mathrm{frp}^2_{\langle 2\rangle}(a,d)$, $\mathrm{frp}^2_{\langle 1,2\rangle}(a,c)$.

But note that, e.g., $\mathrm{frp}^2_{\langle 2,2\rangle}(a,c)$ is not true even though $\langle 2,2\rangle$ is a relative 2-path and it leads to c when starting from a, since so does $\langle 1,2\rangle$, which is lexicographically smaller than $\langle 2,2\rangle$.

...

- We only illustrate the formulas concerning (ℓ,d)-path-based identifiers for $\ell = d = 1$.

  $\mathrm{pid}^{1,1}_{\varepsilon}(a)$.

  $\mathrm{pid}^{1,1}_{\langle\varepsilon\rangle}(a,a)$, $\mathrm{pid}^{1,1}_{\langle\varepsilon\rangle}(b,b)$, $\mathrm{pid}^{1,1}_{\langle\varepsilon\rangle}(c,c)$, $\mathrm{pid}^{1,1}_{\langle\varepsilon\rangle}(d,d)$.

  $\mathrm{pid}^{1,1}_{\langle\varepsilon,\varepsilon\rangle}(a,a,a)$, $\mathrm{pid}^{1,1}_{\langle\varepsilon,\langle 1\rangle\rangle}(a,a,b)$, $\mathrm{pid}^{1,1}_{\langle\langle 1\rangle,\varepsilon\rangle}(d,a,d)$.

  ...

  Even though $b^{\langle\langle 1\rangle,\varepsilon\rangle} = \langle a,b\rangle$, the $(1,1)$-path-based identifier of $\langle a,b\rangle$ is not constituted by b and $\langle\langle 1\rangle,\varepsilon\rangle$ because $a < b$ and there is a tuple $\boldsymbol{\psi}$ of relative $(1,1)$-paths such that $a^{\boldsymbol{\psi}} = \langle a,b\rangle$, namely $\boldsymbol{\psi} = \langle\varepsilon,\langle 1\rangle\rangle$. Moreover, $\langle a,c\rangle$ has no $(1,1)$-path-based identifier because the distance between a and c is 2 and for all elements that are within distance 1 of both a and c (namely b and d), there is no path leading to c that only visits the first neighbor.

  This already suggests a property that we will exploit later: A tuple $\boldsymbol{a}$ of domain elements of $\mathcal{A}$ is guaranteed to have an (ℓ,d)-path-based identifier if the distance between any two elements of $\boldsymbol{a}$ is at most ℓ and the degree of $\mathcal{A}$ is at most d. △

The following statement is rather obvious but crucial:

Lemma 24. *Let Π be a connection-guarded program without constants, let r be a rule in Π and let $\mathcal{A}$ be an input structure of Π. For any two constants a and b in any ground rule $r' \in \mathrm{gr}(\Pi \cup \mathcal{A})$ obtained from r during grounding, the distance between a and b in $\mathcal{A}$ is at most the number of variables in r minus one.*

Proof. If Π is connection-guarded, the distance between any two variables in the join structure of a rule r with n variables is at most $n-1$. By the nature of grounding, a ground instance of r is only produced if there is a homomorphism from the join structure of r to the input structure $\mathcal{A}$. Hence the distance between a and b in $\mathcal{A}$ is at most $n-1$. □

We next show an important property for connection-guarded ASP programs without constants: For each possible input and every tuple $\boldsymbol{a}$ of constants that can occur in an atom of the grounding, $\boldsymbol{a}$ has an (ℓ,d)-path-based identifier.

Lemma 25. *Let Π be a connection-guarded ASP program without constants, ℓ the maximum number of variables in any rule of Π, $\mathcal{A}$ an input structure of Π, d the degree of $\mathcal{A}$, and $p(\boldsymbol{a})$ an atom that occurs in $\mathrm{gr}(\Pi \cup \mathcal{A})$. Then, the tuple $\boldsymbol{a}$ has an (ℓ,d)-path-based identifier in $\mathcal{A}$.*

Proof. By Lemma 24, the distance between any two elements of $\boldsymbol{a} = \langle a_1, \ldots, a_k \rangle$ is at most ℓ, so there is a domain element s such that, for each i, there is a relative (ℓ, d)-path π_i satisfying $s^{\pi_i} = a_i$. As observed when we defined (ℓ, d)-path-based identifiers above, $\boldsymbol{a}$ then has such an identifier in $\mathcal{A}$. □

We can again generalize this as follows.

Lemma 26. *Let* Π *be a connection-guarded ASP program without constants,* ℓ *the maximum number of variables in any rule of* Π*,* $\mathcal{A}$ *an input structure of* Π*,* d *the degree of* $\mathcal{A}$*, and* $p(\boldsymbol{a})$ *and* $q(\boldsymbol{b})$ *atoms that occur together in a rule of* $\mathrm{gr}(\Pi \cup \mathcal{A})$*. Then, the joint tuple* $\boldsymbol{ab}$ *has an* (ℓ, d)*-path-based identifier in* $\mathcal{A}$*.*

Proof. Since $p(\boldsymbol{a})$ and $q(\boldsymbol{b})$ occur together in a rule r of the grounding, the distance between any two elements of $\boldsymbol{ab}$ is at most ℓ by Lemma 24. Hence there is an element s of $\mathrm{dom}(\mathcal{A})$ whose distance to each element of $\boldsymbol{ab}$ is at most ℓ, so there is a relative (ℓ, d)-path from s to each element of $\boldsymbol{ab}$. As observed when we defined (ℓ, d)-path-based identifiers, this means that $\boldsymbol{ab}$ has an (ℓ, d)-path-based identifier in $\mathcal{A}$. □

With the above notions and lemmas established, we are now ready to prove our main theorem in this section.

Proof of Theorem 22. Our proof is similar to that of Theorem 14 for guarded ASP (cf. Figure 5), but here we use the bound d on the degree of input structures in our construction of an appropriate MSO transduction $\kappa_{\Pi,d}$. In this case the transduction thus depends on both Π and the degree bound d, and it is only defined for input structures of degree at most d. Since both Π and d are fixed, we get the desired result by Theorem 5.

The incidence structure $\mathrm{Inc}(\mathcal{A})$ will be an input structure of $\kappa_{\Pi,d}$, and the incidence structure of the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{A})$, which we again consider directed, will be the corresponding output structure. By Theorem 5, this shows our claim. Let ℓ be the maximum number of variables in any rule of Π. Furthermore, assume that $\mathcal{A}$ has bounded treewidth and degree, and let d denote the degree of $\mathcal{A}$. For each rule r of Π, we denote the set of variables of r by $\mathrm{Var}(r)$, and we assume that these variables are denoted by $X_1, \ldots, X_{|\mathrm{Var}(r)|}$. We will, for the moment, assume that Π is constant-free; we will later refer to relevant literature that shows how to handle the general case.

We call the domain elements in the output structure that correspond to vertices and edges of the primal graph *vertex elements* and *edge elements*, respectively. In the following, we define $\kappa_{\Pi,d}$ by the definition scheme $\langle \Delta, \Theta \rangle$, where the tuple Δ is the concatenation of tuples Δ_v and Δ_e, which contain domain formulas that generate the vertex and edge elements, respectively, and Θ contains relation formulas that state which vertex is incident to which edge.[10]

Formulas in Δ_v*.* These formulas shall produce the vertex elements. First we define a formula $\mathrm{inst}_r(\boldsymbol{x})$ for every rule r to express that the tuple $\boldsymbol{x}$ is an instantiation of the

[10] For defining $\kappa_{\Pi,d}$, we choose as the set I, which is used for the subscripts of the formulas, the set containing (a) an element $p[\boldsymbol{\pi}]$ for each predicate p of arity k occurring in Π and for each k-tuple $\boldsymbol{\pi}$ of relative (ℓ, d)-paths, and (b) an element $p[\boldsymbol{\pi}]q[\boldsymbol{\psi}]$ for all predicates p and q, and all tuples $\boldsymbol{\pi}$ and $\boldsymbol{\psi}$ of relative (ℓ, d)-paths, such that the arities of $\boldsymbol{\pi}$ and $\boldsymbol{\psi}$ are the same as those of p and q, respectively. All elements of Δ and Θ that we do not explicitly mention are defined as $\bot$.

variables in r such that the instantiated connection-guard of r indeed appears in the input facts. To this end, let r be a rule in Π. We first define I_r to be the following set of objects: If the connection-guard of r contains an atom $g(X_{i_1}, \ldots, X_{i_k})$, then I_r contains an element $\langle g, i_1, \ldots, i_k \rangle$.

$$\mathrm{inst}_r(x_1, \ldots, x_{|\mathrm{Var}(r)|}) \equiv \bigwedge_{\langle g, i_1, \ldots, i_k \rangle \in I_r} \exists y \big(g(y) \wedge in_1(y, x_{i_1}) \wedge \cdots \wedge in_k(y, x_{i_k}) \big)$$

For convenience, we now define the following formula for all nonnegative integers k and m, and for each tuple $\langle i_1, \ldots, i_k \rangle$ of integers between 1 and m:

$$\mathrm{select}_{\langle i_1, \ldots, i_k \rangle}(x_1, \ldots, x_m, y_1, \ldots, y_k) \equiv y_1 = x_{i_1} \wedge \cdots \wedge y_k = x_{i_k}$$

The formula $\mathrm{select}_{\boldsymbol{i}}(\boldsymbol{x}, \boldsymbol{y})$ is true if and only if the elements of $\boldsymbol{y}$ are those elements of $\boldsymbol{x}$ given by the indices in $\boldsymbol{i}$.

Next, for each predicate p, we define O_p to be the following set of objects: If a rule r in Π contains an atom $p(X_{i_1}, \ldots, X_{i_k})$, then the set O_p contains $\langle r, \boldsymbol{i} \rangle$, where $\boldsymbol{i} = \langle i_1, \ldots, i_k \rangle$. With this, we define a formula $\mathrm{occurs}_p(\boldsymbol{x})$ to express that the ground atom $p(\boldsymbol{x})$ occurs in $\mathrm{gr}(\Pi \cup \mathcal{A})$.

$$\mathrm{occurs}_p(\boldsymbol{x}) \equiv \bigvee_{\langle r, \boldsymbol{i} \rangle \in O_p} \exists \boldsymbol{y} \big(\mathrm{inst}_r(\boldsymbol{y}) \wedge \mathrm{select}_{\boldsymbol{i}}(\boldsymbol{y}, \boldsymbol{x}) \big)$$

We now put this auxiliary formula to use: For each predicate p of arity k occurring in Π and for each k-tuple $\boldsymbol{\pi}$ of relative (ℓ, d)-paths, we define the following formula $\delta_{p[\boldsymbol{\pi}]}(x)$ to be an element of Δ_v.

$$\delta_{p[\boldsymbol{\pi}]}(x) \equiv \exists \boldsymbol{y} \big(\mathrm{pid}^{\ell,d}_{\boldsymbol{\pi}}(x, \boldsymbol{y}) \wedge \mathrm{occurs}_p(\boldsymbol{y}) \big)$$

This formula is true if and only if x together with $\boldsymbol{\pi}$ is the (ℓ, d)-path-based identifier of some tuple $\boldsymbol{a}$ of domain elements and $p(\boldsymbol{a})$ occurs in $\mathrm{gr}(\Pi \cup \mathcal{A})$; the resulting copy of x in the output structure then corresponds to the atom $p(\boldsymbol{a})$.

For each atom $p(\boldsymbol{a})$ in the grounding, we thus produce a vertex element, since $\boldsymbol{a}$ has an (ℓ, d)-path-based identifier by Lemma 25. Moreover, different atoms produce different vertex elements and every vertex element that we produce corresponds to an atom in the grounding by our construction of the occurs_p formulas. This proves that there is a bijection between the atoms in the grounding and the vertex elements.

Formulas in Δ_e. These formulas shall produce the edge elements. We again define an auxiliary formula.

For all predicates p and q of arity k and m, respectively, we define $T_{p,q}$ to be the following set of objects: If there is a rule r in Π such that r contains atoms $p(X_{i_1}, \ldots, X_{i_k})$ and $q(X_{j_1}, \ldots, X_{j_m})$, then $T_{p,q}$ contains $\langle r, \boldsymbol{i}, \boldsymbol{j} \rangle$, where $\boldsymbol{i} = \langle i_1, \ldots, i_k \rangle$ and $\boldsymbol{j} = \langle j_1, \ldots, j_m \rangle$. With this, we now define a formula $\mathrm{together}_{p,q}(\boldsymbol{x}, \boldsymbol{y})$, where $\boldsymbol{x}$ and $\boldsymbol{y}$ are tuples of variables with arity k and m, respectively. This formula expresses that the two ground atoms $p(\boldsymbol{x})$ and $q(\boldsymbol{y})$ occur together in some rule of $\mathrm{gr}(\Pi \cup \mathcal{A})$.

$$\mathrm{together}_{p,q}(\boldsymbol{x}, \boldsymbol{y}) \equiv \bigvee_{\langle r, \boldsymbol{i}, \boldsymbol{j} \rangle \in T_{p,q}} \exists \boldsymbol{z} \big(\mathrm{inst}_r(\boldsymbol{z}) \wedge \mathrm{select}_{\boldsymbol{i}}(\boldsymbol{z}, \boldsymbol{x}) \wedge \mathrm{select}_{\boldsymbol{j}}(\boldsymbol{z}, \boldsymbol{y}) \big)$$

With this auxiliary formula in hand, we define the following formula $\delta_{p[\boldsymbol{\pi}]q[\boldsymbol{\psi}]}(x)$ to be an element of Δ_e, for all predicates p and q, and all tuples $\boldsymbol{\pi}$ and $\boldsymbol{\psi}$ of relative (ℓ, d)-paths, such that the arities of $\boldsymbol{\pi}$ and $\boldsymbol{\psi}$ are the same as those of p and q, respectively.

$$\delta_{p[\boldsymbol{\pi}]q[\boldsymbol{\psi}]}(x) \equiv \begin{cases} \bot & \text{if } p[\boldsymbol{\pi}] = q[\boldsymbol{\psi}] \\ \exists \boldsymbol{y} \exists \boldsymbol{z} \big(\operatorname{pid}^{\ell,d}_{\boldsymbol{\pi}\boldsymbol{\psi}}(x, \boldsymbol{y}, \boldsymbol{z}) \wedge \operatorname{together}_{p,q}(\boldsymbol{y}, \boldsymbol{z}) \big) & \text{otherwise} \end{cases}$$

This formula is true if and only if there are tuples $\boldsymbol{a}$ and $\boldsymbol{b}$ of the same arity as p and q, respectively, such that (1) x together with $\boldsymbol{\pi}\boldsymbol{\psi}$ is the (ℓ, d)-path-based identifier of $\boldsymbol{a}\boldsymbol{b}$, and (2) the atoms $p(\boldsymbol{a})$ and $q(\boldsymbol{b})$ are different and occur together in some rule of $\operatorname{gr}(\Pi \cup \mathcal{A})$. The resulting copy of x in the output structure then corresponds to the edge from $p(\boldsymbol{a})$ to $q(\boldsymbol{b})$ in the primal graph. Due to symmetry, we can see that then also an edge in the other direction will be created. Since both atoms are different if the formula is true, we do not introduce self-loops.

For each pair of different atoms $p(\boldsymbol{a})$ and $q(\boldsymbol{b})$ that jointly occur in a rule of the grounding, we thus correctly produce two edge elements, since $\boldsymbol{a}\boldsymbol{b}$ has an (ℓ, d)-path-based identifier by Lemma 26. Moreover, different such pairs of atoms produce different edge elements, and every edge element that we produce corresponds to a joint occurrence of two atoms in a rule of the grounding by our construction of the $\operatorname{together}_{p,q}$ formulas.

Formulas in Θ. These formulas shall ensure that each edge element is incident to the two appropriate vertex elements. First we define the formula $\operatorname{meet}_{\boldsymbol{\pi},\boldsymbol{\psi}}(x, y)$ for each integer k and all k-tuples $\boldsymbol{\pi}$ and $\boldsymbol{\psi}$ of relative paths. The formula is true if and only if $x^{\boldsymbol{\pi}} = y^{\boldsymbol{\psi}}$.

$$\operatorname{meet}_{\langle \pi_1, \ldots, \pi_k \rangle, \langle \psi_1, \ldots, \psi_k \rangle}(x, y) \equiv \bigwedge_{1 \leq i \leq k} \exists z \big(\operatorname{reach}_{\pi_i}(x, z) \wedge \operatorname{reach}_{\psi_i}(y, z) \big)$$

Now let p and q be predicates occurring in Π, and let $\boldsymbol{\pi}$ and $\boldsymbol{\psi}$ be tuples of relative (ℓ, d)-paths. We define a formula $\operatorname{eq}_{p[\boldsymbol{\pi}],q[\boldsymbol{\psi}]}(x, y)$ to express that the atoms $p(x^{\boldsymbol{\pi}})$ and $q(y^{\boldsymbol{\psi}})$ are equal.

$$\operatorname{eq}_{p[\boldsymbol{\pi}],q[\boldsymbol{\psi}]}(x, y) \equiv \begin{cases} \operatorname{meet}_{\boldsymbol{\pi},\boldsymbol{\psi}}(x, y) & \text{if } p = q \\ \bot & \text{otherwise} \end{cases}$$

Let p, q and q' be predicates occurring in Π, and let $\boldsymbol{\pi}$, $\boldsymbol{\psi}$ and $\boldsymbol{\psi}'$ be tuples of relative (ℓ, d)-paths with the same arity as p, q and q', respectively. We define the following formulas to be an element of Θ:

$$\begin{aligned} \theta_{in_1,\, p[\boldsymbol{\pi}],\, q[\boldsymbol{\psi}]q'[\boldsymbol{\psi}']}(x, y) &\equiv \operatorname{eq}_{p[\boldsymbol{\pi}],q[\boldsymbol{\psi}]}(x, y) \\ \theta_{in_2,\, p[\boldsymbol{\pi}],\, q[\boldsymbol{\psi}]q'[\boldsymbol{\psi}']}(x, y) &\equiv \operatorname{eq}_{p[\boldsymbol{\pi}],q'[\boldsymbol{\psi}']}(x, y) \end{aligned}$$

We only explain the first of these formulas, as the other case is symmetric.

The formula $\theta_{in_1,\, p[\boldsymbol{\pi}],\, q[\boldsymbol{\psi}]q'[\boldsymbol{\psi}']}(x, y)$ is true if and only if the atom $p(\boldsymbol{a})$ is equal to $q(\boldsymbol{b})$, where $\boldsymbol{a} = x^{\boldsymbol{\pi}}$ and $\boldsymbol{b} = y^{\boldsymbol{\psi}}$. If this formula is true, it makes the edge represented by the respective copy of y an *outgoing* edge of $p(\boldsymbol{a})$ because of the subscript in_1.

We first show that, whenever our transduction causes an edge element to be incident to a vertex element, the corresponding edge in the primal graph is indeed an outgoing edge of the appropriate vertex. Suppose there are predicates p, q and p', tuples $\boldsymbol{\pi}$, $\boldsymbol{\psi}$ and $\boldsymbol{\psi}'$,

as well as domain elements x and y such that (1) $\delta_{p[\boldsymbol{\pi}]}(x)$ is true, (2) $\delta_{q[\boldsymbol{\psi}]q'[\boldsymbol{\psi}']}(y)$ is true, and (3) $\theta_{in_1,\, p[\boldsymbol{\pi}],\, q[\boldsymbol{\psi}]q'[\boldsymbol{\psi}']}(x,y)$ is true. As observed in our definition of Δ_v, (1) means that x together with $\boldsymbol{\pi}$ is the (ℓ,d)-path-based identifier of some tuple $\boldsymbol{a}$, and the grounding contains an atom $p(\boldsymbol{a})$. From (2) we get that there are tuples $\boldsymbol{b}$ and $\boldsymbol{b}'$ such that y together with $\boldsymbol{\psi}\boldsymbol{\psi}'$ is the (ℓ,d)-path-based identifier of $\boldsymbol{b}\boldsymbol{b}'$. We have also seen that by (2) there is a rule in the grounding that contains both $q(\boldsymbol{b})$ and $q'(\boldsymbol{b}')$, hence there is an edge from $q(\boldsymbol{b})$ to $q'(\boldsymbol{b}')$ in the primal graph. By (3) and the definition of (ℓ,d)-path-based identifiers (in particular those of $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{b}'$), we know that $p(\boldsymbol{a})$ is equal to $q(\boldsymbol{b})$. Hence the edge in the primal graph from $q(\boldsymbol{b})$ to $q'(\boldsymbol{b}')$ is indeed an outgoing edge of $p(\boldsymbol{a})$.

We now prove the other direction: Whenever an edge in the primal graph is an outgoing edge of a vertex, a formula in Θ defining the relation in_1 causes the corresponding edge element to an outgoing edge of the appropriate vertex element. Suppose that the primal graph contains an edge from atom $p(\boldsymbol{a})$ to atom $q(\boldsymbol{b})$. Then these atoms occur together in a rule of the grounding and, as we have seen in our definition of Δ_v and Δ_e, our transduction produces a vertex element v for $p(\boldsymbol{a})$ and an edge element e for the edge from $p(\boldsymbol{a})$ to $q(\boldsymbol{b})$. Now let x and y be domain elements of $\mathcal{A}$, and let $\boldsymbol{\pi}$, $\boldsymbol{\psi}$ and $\boldsymbol{\psi}'$ be tuples of relative (ℓ,d)-paths, such that x together with $\boldsymbol{\pi}$ is the (ℓ,d)-path-based identifier of $\boldsymbol{a}$, and y together with $\boldsymbol{\psi}\boldsymbol{\psi}'$ is the (ℓ,d)-path-based identifier of $\boldsymbol{a}\boldsymbol{b}$. By definition of (ℓ,d)-path-based identifiers, $x^{\boldsymbol{\pi}} = \boldsymbol{a}$; moreover, $y^{\boldsymbol{\psi}\boldsymbol{\psi}'} = \boldsymbol{a}\boldsymbol{b}$, which entails $y^{\boldsymbol{\psi}} = \boldsymbol{a}$. Since both $\boldsymbol{a} = x^{\boldsymbol{\pi}}$ and $\boldsymbol{a} = y^{\boldsymbol{\psi}}$, the formula $\mathrm{eq}_{p[\boldsymbol{\pi}],p[\boldsymbol{\psi}]}(x,y)$ is clearly true. Hence the formula $\theta_{in_1,\, p[\boldsymbol{\pi}],\, p[\boldsymbol{\psi}]q[\boldsymbol{\psi}']}(x,y)$ is true, which correctly makes the edge element e an outgoing edge of the vertex element v.

For the remaining relation formulas, which define the relation E, we proceed in the same way as in the proof of Theorem 14.

This completes the construction of the MSO transduction $\kappa_{\Pi,d}$. Let $\mathcal{A}$ be an input structure for Π with degree bounded by d. Clearly, since Π, d and ℓ are fixed, so is $\kappa_{\Pi,d}$. We have argued that $\kappa_{\Pi,d}(\mathrm{Inc}(\mathcal{A}))$ yields the incidence structure of the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{A})$ as desired. By Theorem 5 and the fact that $\mathcal{A}$, by assumption, has bounded treewidth, this proves our claim for constant-free programs.

We can also generalize this proof to programs with constants (Bliem, 2017). □

We now illustrate this transduction for the same program as in Example 18.

Example 27. Let Π be the connection-guarded program from Example 18. We denote the rules by $r_1, \ldots, r_4$ from top to bottom. Again let $\mathcal{G}$ be the input structure for Π satisfying $\mathrm{dom}(\mathcal{G}) = \{a, b\}$, $succ^{\mathcal{G}} = \{\langle a, b\rangle\}$ and $\mathtt{e}^{\mathcal{G}} = \{\langle b, a\rangle\}$, and whose degree is bounded by some integer d. We denote the domain element of $\mathrm{Inc}(\mathcal{G})$ for the fact $\langle b, a\rangle$ in $\mathtt{e}^{\mathcal{G}}$ and for $\langle a, b\rangle$ in $succ^{\mathcal{G}}$ by ba and ab, respectively, and we assume the ordering $a < b < ab < ba$. We show how $\kappa_{\Pi,d}$ transforms $\mathrm{Inc}(\mathcal{G})$ into the incidence structure of the primal graph of $\mathrm{gr}(\Pi \cup \mathcal{G})$.

For the vertex elements, first observe that the formula $\mathrm{inst}_r(b, a)$ is true for every rule r because each rule has the guard $\mathtt{e}(\mathtt{X}, \mathtt{Y})$ and the only input fact is $\mathtt{e}(b, a)$. Now observe that all of the formulas $\mathrm{occurs}_{\mathtt{e}}(b, a)$, $\mathrm{occurs}_{\mathtt{r}}(a)$, $\mathrm{occurs}_{\mathtt{r}}(b)$, $\mathrm{occurs}_{\mathtt{g}}(a)$ and $\mathrm{occurs}_{\mathtt{g}}(b)$ are true. For instance, observe that $O_{\mathtt{e}} = \{\langle r_1, 1, 2\rangle, \langle r_2, 1, 2\rangle, \langle r_3, 1, 2\rangle, \langle r_4, 1, 2\rangle\}$ (under the assumption that the first variable of each rule is $\mathtt{X}$ and the second is $\mathtt{Y}$); now clearly the subformula $\mathrm{inst}_r(b, a) \wedge \mathrm{select}_{\langle 1,2\rangle}(b, a, b, a)$ is true for every rule r. Hence $\mathrm{occurs}_{\mathtt{e}}(b, a)$ is true. Since the $(2, d)$-path-based identifier of $\langle b, a\rangle$ is b in combination with $\langle \varepsilon, \langle 1\rangle\rangle$, the formula $\mathrm{pid}^{2,d}_{\langle \varepsilon, \langle 1\rangle\rangle}(b, b, a)$ is true. It is now easy to verify that $\delta_{\mathtt{e}[\varepsilon, \langle 1\rangle]}(b)$ is true by looking at

its definition

$$\delta_{\mathtt{e}[\varepsilon,\langle 1\rangle]}(b) \;\equiv\; \exists y_1 \exists y_2 \big(\mathrm{pid}^{2,d}_{\langle\varepsilon,\langle 1\rangle\rangle}(b, y_1, y_2) \wedge \mathrm{occurs}_{\mathtt{e}}(y_1, y_2)\big).$$

Thus we produce the vertex element for $\mathtt{e}(b, a)$.

Similarly, $O_{\mathtt{r}} = \{\langle r_1, 1\rangle, \langle r_2, 2\rangle, \langle r_3, 1\rangle, \langle r_3, 2\rangle\}$, by which we can see that both $\mathrm{occurs}_{\mathtt{r}}(a)$ and $\mathrm{occurs}_{\mathtt{r}}(b)$ are true due to the elements of $O_{\mathtt{r}}$ containing 2 and 1, respectively. The $(2, d)$-path-based identifiers of $\langle a\rangle$ and $\langle b\rangle$ are a together with $\langle\varepsilon\rangle$ and b together with $\langle\varepsilon\rangle$, respectively. This makes $\delta_{\mathtt{r}[\varepsilon]}(a)$ and $\delta_{\mathtt{r}[\varepsilon]}(b)$ true and produces the vertex elements for $\mathtt{r}(a)$ and $\mathtt{r}(b)$, respectively.

We illustrate the edge elements just by the construction for the edge from atom $\mathtt{r}(a)$ to atom $\mathtt{e}(b, a)$. Note that $\mathrm{together}_{\mathtt{r},\mathtt{e}}(\langle a\rangle, \langle b, a\rangle)$ is true because $T_{\mathtt{r},\mathtt{e}}$ contains the tuple $\langle r_1, \langle 2\rangle, \langle 1, 2\rangle\rangle$ as well as $\langle r_3, \langle 2\rangle, \langle 1, 2\rangle\rangle$, and clearly both subformulas $\mathrm{inst}_{r_1}(b, a) \wedge \mathrm{select}_{\langle 2\rangle}(b, a, a) \wedge \mathrm{select}_{\langle 1,2\rangle}(b, a, b, a)$ and $\mathrm{inst}_{r_3}(b, a) \wedge \mathrm{select}_{\langle 2\rangle}(b, a, a) \wedge \mathrm{select}_{\langle 1,2\rangle}(b, a, b, a)$ are true. Moreover, observe that b together with $\langle\langle 1\rangle, \varepsilon, \langle 1\rangle\rangle$ is the $(2, d)$-path-based identifier of $\langle a, b, a\rangle$. So the formula $\mathrm{pid}^{2,d}_{\langle\langle 1\rangle,\varepsilon,\langle 1\rangle\rangle}(b, a, b, a)$ is true. Hence, by looking at the definition

$$\delta_{\mathtt{r}[\langle 1\rangle]\mathtt{e}[\varepsilon,\langle 1\rangle]}(b) \;\equiv\; \exists y_1 \exists z_1 \exists z_2 \big(\mathrm{pid}^{2,d}_{\langle\langle 1\rangle,\varepsilon,\langle 1\rangle\rangle}(b, y_1, z_1, z_2) \wedge \mathrm{together}_{\mathtt{r},\mathtt{e}}(\langle y_1\rangle, \langle z_1, z_2\rangle)\big),$$

we can conclude that $\delta_{\mathtt{r}[\langle 1\rangle]\mathtt{e}[\varepsilon,\langle 1\rangle]}(b)$ is true, which produces the desired edge element.

Finally we show that our transduction indeed makes the edge that should go from $\mathtt{r}(a)$ to $\mathtt{e}(b, a)$ an outgoing edge of $\mathtt{r}(a)$. Recall that the vertex element exists due to $\delta_{\mathtt{r}[\varepsilon]}(a)$ being true, and the edge element exists due to $\delta_{\mathtt{r}[\langle 1\rangle]\mathtt{e}[\varepsilon,\langle 1\rangle]}(b)$ being true. Looking at the definition

$$\theta_{in_1,\,\mathtt{r}[\varepsilon],\,\mathtt{r}[\langle 1\rangle]\mathtt{e}[\varepsilon,\langle 1\rangle]}(a, b) \;\equiv\; \mathrm{eq}_{\mathtt{r}[\varepsilon],\mathtt{r}[\langle 1\rangle]}(a, b),$$

we can observe that clearly $\mathrm{eq}_{\mathtt{r}[\varepsilon],\mathtt{r}[\langle 1\rangle]}(a, b)$ is true, so the vertex is incident to the edge element as desired. △

Obviously every guarded ASP program is also connection-guarded. The other direction, however, is not true. Consider, for example, the connection-guarded program consisting of the rule $\mathtt{p}(\mathtt{X}, \mathtt{Z}) \leftarrow \mathtt{e}(\mathtt{X}, \mathtt{Y}),\ \mathtt{e}(\mathtt{Y}, \mathtt{Z})$. As an input for this program, consider a path of length 2. The atoms over predicate $\mathtt{p}$ in the unique answer set contains both endpoints of the path. However, there can be no equivalent guarded program since answer sets of groundings resulting from guarded programs have the property that the atoms over any predicate can only be a subset of the atoms over an extensional predicate.

While connection-guarded ASP is a strict superset of guarded ASP in the sense that each guarded program is connection-guarded but not vice versa, there seems to be a price to pay for the higher generality when the objective is to keep the treewidth of the grounding low: Comparing Theorem 14, which concerns guarded ASP programs, with Theorem 22, which concerns connection-guarded programs, we notice that we rely on the input having bounded degree for showing that grounding connection-guarded programs preserves bounded treewidth of the input, whereas there is no such assumption for guarded programs.

It is natural to ask whether this additional condition is necessary for connection-guarded programs. Unfortunately it is, as witnessed by the rule $\mathtt{p}(\mathtt{X}, \mathtt{Z}) \leftarrow \mathtt{e}(\mathtt{X}, \mathtt{Y}),\ \mathtt{e}(\mathtt{Y}, \mathtt{Z})$, where $\mathtt{e}$ is extensional: When given a tree of height 1 with n vertices (and thus of treewidth 1 and

maximum degree $n-1$), the primal graph of the grounding has linear treewidth, as the complete bipartite graph $K_{n-1,n-1}$ is a subgraph of it.

Also the restrictions in Definition 21 cannot easily be relaxed without destroying bounded treewidth already with very simple programs: If we allow "unconnected" rules like $\mathtt{p}(\mathtt{X},\mathtt{Y}) \leftarrow \mathtt{v}(\mathtt{X}),\ \mathtt{v}(\mathtt{Y})$, then the complete graph K_n is a subgraph of the primal graph of the grounding for any n-vertex instance.

Moreover, if we change the definition of the extensional join graph by drawing edges also for intensional atoms, then the first encoding in Example 1 is allowed, which generates K_n for any connected graph.

5. Complexity

We have seen that the restrictions imposed by the class of connection-guarded ASP are optimal in the sense that removing any one of them destroys the desired property that grounding preserves bounded treewidth of the input when the degree is also bounded. On the other hand, guarded ASP does not require the bound on the degree, but it is syntactically even more restrictive. This raises suspicions about whether the classes are too restrictive to be useful.

Luckily, straightforward encodings for problems like Graph Coloring or Hamiltonian Cycle even fall into the class of guarded ASP (cf. the second encoding in Example 1), and consequently also into connection-guarded ASP. Moreover, it turns out that the restrictions imposed by guardedness and connection-guardedness do not alleviate the complexity of deciding answer set existence when the program is fixed:

Theorem 28. *It is Σ_2^{P}-complete to decide for a fixed guarded or connection-guarded ASP program Π and a given input structure $\mathcal{A}$ whether $\Pi \cup \mathcal{A}$ has an answer set.*

Proof. Membership follows from the general case. For hardness, we present a guarded encoding for the well-known Σ_2^{P}-complete problem $\textsc{Qsat}_2$. We are given a formula

$$\exists x_1 \cdots \exists x_k \forall y_1 \cdots \forall y_\ell \, \varphi,$$

where φ is a formula in 3-DNF (i.e., a disjunction of conjunctive terms, each containing at most three literals), and the question is whether there are truth values for the x variables such that for all truth values for the y variables φ is true. We assume that each disjunct in φ contains exactly three literals, which can be achieved by using the same literal multiple times in a disjunct.

Consider the ASP program in Figure 6, which is based on the encoding in Section 3.3.5 of Leone et al. (2006). The $\textsc{Qsat}_2$ formula is represented as a structure $\mathcal{A}$ as follows: The domain of $\mathcal{A}$ consists of all variables in φ and two special elements $\top$ and $\bot$. We choose $\mathtt{verum}^{\mathcal{A}} = \{\top\}$ and $\mathtt{falsum}^{\mathcal{A}} = \{\bot\}$. The relations $\mathtt{exists}^{\mathcal{A}}$ and $\mathtt{forall}^{\mathcal{A}}$ consist of all existentially and universally quantified variables, respectively. Finally, for each disjunct $l_1 \wedge l_2 \wedge l_3$ in the formula, we put an element $\langle p_1, p_2, p_3, q_1, q_2, q_3 \rangle$ into $\mathtt{term}^{\mathcal{A}}$, where p_i denotes v_i if l_i is a positive atom v_i, otherwise $p_i = \top$, and q_i denotes v_i if l_i is an atom of the form $\mathtt{not}\ v_i$, otherwise $q_i = \bot$. The element $\langle p_1, p_2, p_3, q_1, q_2, q_3 \rangle$ thus represents $p_1 \wedge p_2 \wedge p_3 \wedge \neg q_1 \wedge \neg q_2 \wedge \neg q_3$, which is equivalent to the original disjunct. This program is

```
      t(T) ← verum(T).
      f(F) ← falsum(F).
t(X) ∨ f(X) ← exists(X).
t(Y) ∨ f(Y) ← forall(Y).
        w ← term(X,Y,Z,Na,Nb,Nc), t(X), t(Y), t(Z), f(Na), f(Nb), f(Nc).
      t(Y) ← w, forall(Y).
      f(Y) ← w, forall(Y).
         ← not w.
```

Figure 6: An encoding of QSAT_2 in guarded ASP

	treewidth	degree	treewidth + degree
guarded	FPT (29)	Σ_2^{P}-complete (32)	FPT
connection-guarded	NP-hard (31)	Σ_2^{P}-complete	FPT (30)

Table 1: Parameterized complexity of answer set existence for our considered classes when the program is fixed. In parentheses: Number of the theorem proving the result. Results without parentheses are implied by other results.

clearly guarded and indeed encodes the QSAT_2 problem, as can be seen by the arguments in Leone et al. (2006). □

Having concluded our investigation of the classical complexity of the answer set existence problem for our classes, we now turn to the parameterized complexity of solving fixed guarded and connection-guarded ASP programs. The parameters we consider are the treewidth and the degree of the input structures, as well as the combined parameter treewidth + degree. Our most important results are summarized in Table 1. In the following we prove these results (and more) individually.

When we consider the treewidth of the input structures as the parameter, we will see that answer-set solving for fixed guarded programs is in fact fixed-parameter tractable. It is also fixed-parameter tractable if we consider fixed connection-guarded programs and the combination of treewidth and degree as the parameter. In other words, the ASP classes we defined are not only useful for encoding problems in such a way that we implicitly benefit from the treewidth-sensitivity inherent to state-of-the-art ASP solvers due to the results in Theorem 14 and Theorem 22, but they are also amenable to algorithms that explicitly exploit small treewidth.

Theorem 29. *For every fixed guarded ASP program* Π *and every family* $\mathbb{A}$ *of input structures of bounded treewidth, the problem of deciding whether, given any input structure* $\mathcal{A} \in \mathbb{A}$*, the program* $\Pi \cup \mathcal{A}$ *has an answer set can be decided in linear time.*

Proof. As we have shown in Theorem 14, grounding a fixed guarded ASP program Π together with an input structure $\mathcal{A}$ leads to a grounding Γ whose treewidth only depends on the treewidth of $\mathcal{A}$. The size of the grounding is linear in the size of $\mathcal{A}$, since every rule in Π is guarded and thus has at most one ground instance in Γ for every fact in $\mathcal{A}$. We can now simply use Γ as the input for a fixed-parameter linear algorithm solving ground ASP. Gottlob et al. (2010) showed that such an algorithm exists. □

Unsurprisingly, an analogous statement can be shown for connection-guarded ASP.

Theorem 30. *For every fixed connection-guarded ASP program* Π *the problem of deciding for a given input structure* $\mathcal{A}$ *whether* $\Pi \cup \mathcal{A}$ *has an answer set is fixed-parameter linear when parameterized by the combination of the treewidth and degree of* $\mathcal{A}$*.*

Proof. Since Π is connection-guarded, observe that the size of $\mathrm{gr}(\Pi \cup \mathcal{A})$ is in $\mathcal{O}(n \cdot d^{\ell})$, where n and d denote the size and degree of $\mathcal{A}$, respectively, and ℓ is the maximum number of variables in a rule of Π. Hence, for bounded d and ℓ, the size of the grounding is linear in n. We can now prove the statement in the same way as Theorem 29, with the modification that we invoke Theorem 22 instead of Theorem 14. □

We obtained our positive results on connection-guarded ASP by parameterizing the problem by the combination of treewidth and degree, whereas guarded ASP for any fixed non-ground encoding is already FPT when parameterized by treewidth only. It is thus natural to ask whether, for fixed encodings, connection-guarded ASP is FPT when parameterized only by either treewidth or degree.

Recall that in Section 4.3 we pointed out that grounding a connection-guarded encoding together with an input structure of arbitrary degree may lead to unbounded treewidth of the grounding. It is therefore not very surprising that the degree bound is indeed necessary for obtaining fixed-parameter tractability (unless $\mathsf{P} = \mathsf{NP}$):

Theorem 31. *The problem of deciding whether a fixed connection-guarded program* Π *together with a given input structure* $\mathcal{A}$ *has an answer set is* NP*-hard. This even holds if the treewidth of* $\mathcal{A}$ *is at most three and* Π *contains no disjunctions.*

Proof. We reduce from the following NP-complete problem.

> Subgraph Isomorphism
> Input: Graphs G and H
> Question: Is there a subgraph of G that is isomorphic to H?

This problem remains NP-hard even if the treewidth of both G and H is at most two (Matoušek & Thomas, 1992).

Let $\langle G, H \rangle$ be an instance of Subgraph Isomorphism. We will present a connection-guarded ASP encoding for the Subgraph Isomorphism problem using the signature $\sigma = \{\mathtt{vg}, \mathtt{vh}, \mathtt{eg}, \mathtt{eh}, \mathtt{bridge}, \mathtt{eq}\}$, where `vg` and `vh` are unary predicates used to represent the vertices of G and H, respectively; `eg` and `eh` are binary predicates for the respective edges; the binary predicate `bridge` is used to connect each vertex of G with a new "bridge element", which is in turn connected to each vertex of H also via the `bridge` predicate; and the binary `eq` predicate contains all pairs of equal vertices. According to this intended meaning, we define a structure $\mathcal{A}$ over σ by $\mathrm{dom}(\mathcal{A}) = \mathrm{V}(G) \cup \mathrm{V}(H) \cup \{\mathtt{b}\}$ (where `b` is a new element), $\mathtt{vg}^{\mathcal{A}} = \mathrm{V}(G)$, $\mathtt{vh}^{\mathcal{A}} = \mathrm{V}(H)$, $\mathtt{eg}^{\mathcal{A}} = \mathrm{E}(G)$, $\mathtt{eh}^{\mathcal{A}} = \mathrm{E}(H)$, $\mathtt{bridge}^{\mathcal{A}} = \{(g, \mathtt{b}), (\mathtt{b}, h) \mid g \in \mathrm{V}(G),\ h \in \mathrm{V}(H)\}$ and $\mathtt{eq}^{\mathcal{A}} = \{(v, v) \mid v \in \mathrm{V}(G) \cup \mathrm{V}(H)\}$. Note that for every pair (x, y) in $\mathtt{eg}^{\mathcal{A}}$ or $\mathtt{eh}^{\mathcal{A}}$ there is also (y, x) in $\mathtt{eg}^{\mathcal{A}}$ or $\mathtt{eh}^{\mathcal{A}}$, respectively, since the graphs are undirected.

```
% Guess a subgraph S of G using predicates vs/1 and es/2.
      vs(X) ← vg(X), not not_vs(X).
  not_vs(X) ← vg(X), not vs(X).
    es(X,Y) ← eg(X,Y), vs(X), vs(Y), not not_es(X,Y).
not_es(X,Y) ← eg(X,Y), vs(X), vs(Y), not es(X,Y).
% Guess a relation representing an isomorphism using predicate iso/2.
    iso(G,H) ← vs(G), vh(H), not not_iso(G,H), bridge(G,B), bridge(B,H).
not_iso(G,H) ← vs(G), vh(H), not iso(G,H), bridge(G,B), bridge(B,H).
% The guessed relation must be a bijection from V(S) to V(H).
        ← iso(G,H1), iso(G,H2), not eq(H1,H2),
          bridge(G,B), bridge(B,H1), bridge(B,H2).
        ← iso(G1,H), iso(G2,H), not eq(G1,G2),
          bridge(G1,B), bridge(G2,B), bridge(B,H).
used(G) ← iso(G,H), bridge(G,B), bridge(B,H).
used(H) ← iso(G,H), bridge(G,B), bridge(B,H).
        ← vg(G), vs(G), not used(G).
        ← vh(H), not used(H).
% The guessed relation must be an isomorphism.
← iso(G1,H1), iso(G2,H2), es(G1,G2), not eh(H1,H2),
  bridge(G1,B), bridge(G2,B), bridge(B,H1), bridge(B,H2).
← iso(G1,H1), iso(G2,H2), eh(H1,H2), not es(G1,G2),
  bridge(G1,B), bridge(G2,B), bridge(B,H1), bridge(B,H2).
```

Figure 7: An encoding of Subgraph Isomorphism in connection-guarded ASP

The connection-guarded program in Figure 7 encodes Subgraph Isomorphism.[11] By construction, H is isomorphic to a subgraph of G if and only if $\Pi \cup \mathcal{A}$ has an answer set.

The treewidth of $\mathcal{A}$ is the maximum of the treewidth of G and of H plus one: Given tree decompositions $\mathcal{T}_G$ and $\mathcal{T}_H$ of G and H, respectively, we can obtain a tree decomposition of $\mathcal{A}$ by taking the disjoint union of $\mathcal{T}_G$ and $\mathcal{T}_H$, adding the bridge element $\mathtt{b}$ to every bag and drawing an edge between an arbitrary node from $\mathcal{T}_G$ and an arbitrary node from $\mathcal{T}_H$. □

ASP solving for connection-guarded programs thus remains hard if only the treewidth of the input is bounded. In fact our result not only rules out fixed-parameter tractable algorithms but even polynomial-time algorithms when we consider the treewidth of the input as a constant (unless $\mathsf{P} = \mathsf{NP}$).

If we allow disjunctions, answer set existence for fixed programs becomes Σ_2^P-complete in general, whereas we have shown in Theorem 31 that the problem is NP-hard when the program is connection-guarded and the input has bounded treewidth. Since this is only a hardness result, there might still be hope for disjunctive ASP that bounded treewidth lowers the complexity by one level of the polynomial hierarchy. However, we consider this unlikely and we suspect that answer set existence for fixed connection-guarded programs is Σ_2^P-complete for bounded treewidth. Since we are not aware of any problems that have been shown to be complete for this class on instances of bounded treewidth, we leave this as an open question.

We now prove that the degree alone is also not sufficient for obtaining fixed-parameter tractability, even if the fixed program is guarded.

The following statement says that it is not enough to consider the degree alone as a parameter. Indeed, even for guarded programs, the complexity of answer set existence remains as hard under this assumption as it is in the case of general ASP.

Theorem 32. *It is Σ_2^P-complete to decide for a fixed guarded program Π and a given input structure $\mathcal{A}$ whether $\Pi \cup \mathcal{A}$ has an answer set even if the degree of $\mathcal{A}$ is at most 15.*

Proof. Membership follows from the general case. For hardness, we present a reduction from the well-known Σ_2^P-complete problem QSAT_2. We are given a formula $\exists x_1 \cdots \exists x_k \forall y_1 \cdots \forall y_\ell \, \varphi$, where φ is a formula in 3-DNF, and the question is whether there are truth values for the x variables such that for all truth values for the y variables φ is true. We may assume that every variable occurs at most three times in φ (Peters, 2017).

We can use the same ASP encoding as in the proof of Theorem 28, but we need to choose a slightly different input structure because the domain elements $\top$ and $\bot$ from that construction have unbounded degree. Recall that the old construction puts an element $\langle p_1, p_2, p_3, q_1, q_2, q_3 \rangle$ into $\mathtt{term}^\mathcal{A}$ for each disjunct in φ and that some p_i or q_j may be $\top$ or $\bot$ in order to represent the equivalent term $p_1 \wedge p_2 \wedge p_3 \wedge \neg q_1 \wedge \neg q_2 \wedge \neg q_2$. The only thing that matters for $\top$ and $\bot$ is that they are always interpreted as true and false, respectively, which the old construction ensures by putting them in $\mathtt{verum}^\mathcal{A}$ and $\mathtt{falsum}^\mathcal{A}$, respectively. We can thus just use a certain number of copies of $\top$ and $\bot$ such that every copy occurs

[11] In practice, we could simplify this encoding substantially by using convenient language constructs provided by ASP systems. For the purpose of this proof, we use our rather restrictive base language. Moreover, note that the positive body of many rules contains atoms whose only purpose is to make the rules connection-guarded. Such redundant atoms could be omitted in practice.

exactly once in $\texttt{term}^{\mathcal{A}}$ and every copy is in the respective $\texttt{verum}^{\mathcal{A}}$ or $\texttt{falsum}^{\mathcal{A}}$ relation. Clearly this reduction to ASP is still correct. The degree of $\mathcal{A}$ is at most 15 because every domain element has at most five neighbors in each tuple of $\texttt{term}^{\mathcal{A}}$ and every variable occurs in at most three tuples. □

6. Discussion

In our investigation of the effect of grounding on the treewidth, we rely on the rather primitive notion of grounding from Definition 6. State-of-the-art grounders, on the other hand, produce groundings whose primal graphs are generally subgraphs of the output of our transductions. However, since degree and treewidth of a graph can only decrease for a subgraph, our results apply also to state-of-the-art grounders.

Moreover, state-of-the-art grounders are capable of solving problems without needing to call an ASP solver if the program has an answer set that is a deterministic consequence of the input, i.e., if no non-deterministic guessing is involved. This is the case, for instance, for Horn programs (that is, ASP programs without negation, disjunction and aggregates). Our notion of grounding, on the other hand, assumes that the grounder does not propagate deterministic consequences and thus cannot solve such simple problems by itself. This is in fact a reasonable assumption: The question we are concerned with in this work is which form a non-ground rule may have so it does not "destroy" bounded treewidth of the input. Any encoding can be made "nasty" in the sense that a grounder cannot solve the problem, namely by forcing atoms to be guessed. This prevents the grounder from eliminating atoms from rule bodies, and it does not change the form of rules.

There have been some investigations concerning treewidth in the context of ASP. Beside parameterized complexity results (Gottlob et al., 2010; Pichler et al., 2014) for ground programs, there was also work on tree-decomposition-based dynamic programming algorithms (Jakl et al., 2009) and their implementations (Morak, Pichler, Rümmele, & Woltran, 2010; Fichte et al., 2017). However, in contrast to the current work, most studies of treewidth in ASP solving only considered the ground case.

Tree decompositions have been applied in the context of non-ground ASP for rule decomposition techniques (Bichler, Morak, & Woltran, 2017, 2016). The goal of this, however, is improving efficiency without explicitly aiming at fixed-parameter tractability. Hence those efforts go in a different direction than the current work.

The FPT result in Theorem 29 (or Theorem 30) has the side effect that (connection-) guarded ASP can also serve as a tool for establishing that a problem is fixed-parameter tractable when parameterized by treewidth (or by the combination of treewidth and degree). Using our ASP classes in this way is similar to a very established technique for classifying a problem parameterized by treewidth as FPT: If the problem can be expressed in MSO, then by Courcelle's theorem the FPT membership follows. It is therefore a natural question how our ASP classes relate to MSO.

It is well known that MSO model checking, just as first-order model checking, is PSPACE-complete (Stockmeyer, 1974; Vardi, 1982), and it is PSPACE-hard even if the input structure is fixed and contains only two domain elements (Kreutzer, 2012). In contrast, we know that we cannot express problems harder than Σ_2^{P} in ASP unless the polynomial hierarchy

collapses to the second level. Hence there are problems that can be expressed using MSO but not in our ASP classes.

Nevertheless, there are also problems that can be expressed in connection-guarded ASP but not in MSO. As we have seen in Theorem 31, answer set existence for fixed connection-guarded programs is NP-hard for instances of bounded treewidth. But by Courcelle's theorem, a reduction to MSO would imply FPT membership.

Of course, the fact that connection-guarded ASP allows us to define some problems that we cannot define with MSO is only of limited significance because connection-guarded ASP allows us to obtain FPT results when the parameter is the combination of treewidth and degree, whereas we only need treewidth as the parameter for an FPT result using MSO. Still, the class of connection-guarded programs may be of interest for algorithmic purposes because it allows us to classify a problem as FPT when parameterized by treewidth plus degree, as we have shown in Theorem 30. We are not aware of any extensions of MSO that allow us to obtain new FPT results using this more restrictive parameter. Hence our result may lead to an extension of MSO that can be used for classifying problems as FPT when the parameter is treewidth + degree. This is subject of future work.

MSO has been extended in several ways to increase its expressive power while retaining FPT (or at least XP) membership of model checking when parameterized by treewidth. For overviews, we refer the reader to Knop, Koutecký, Masarík, and Toufar (2017) and Langer, Reidl, Rossmanith, and Sikdar (2014). On the other hand, also our results on (connection-) guarded ASP can be extended to more powerful languages. This has, in fact, been done by Bliem (2017) for weak constraints and aggregates. We believe that especially connection-guarded ASP with aggregates and weak constraints is an attractive language because it allows us to express interesting problems and FPT classification tools for the parameter treewidth + degree seem to be rare.

Even though in many cases a problem can also be expressed in (an extension of) MSO, this is mostly interesting from a theoretical perspective, whereas the actual solving performance of algorithms based on MSO model checking is usually clearly worse than that of dedicated tools (Cygan, Fomin, Kowalik, Lokshtanov, Marx, Pilipczuk, Pilipczuk, & Saurabh, 2015, pp. 184–185), even though there have been considerable advances in this effort (Langer, Reidl, Rossmanith, & Sikdar, 2012; Bannach & Berndt, 2018). Even for rather simple problems, MSO formulas can unfortunately be quite complex. For many problems in Σ_2^{P}, expressing a problem in one of our classes not only yields a result about its complexity by Theorems 29 and 30, but it should in most cases also give quite good performance in practice due to the efficiency of ASP systems.

7. Conclusions

In this paper, we experimentally showed that modern ASP solvers perform better when the ground input programs have small treewidth, all other things being equal. This is strong evidence that one should not only aim for small groundings when encoding problems in ASP, but also for groundings of small treewidth.

With this observation in mind, we identified two classes of ASP programs, namely guarded and connection-guarded programs, with favorable properties. In particular, we could show that guarded programs yield groundings of small treewidth whenever a graph

representation of the input facts has small treewidth. For connection-guarded programs, the same holds if additionally also the maximum degree of this graph is small.

Furthermore, we investigated the complexity of relevant reasoning problems on these classes of programs. While guarded and connection-guarded programs have a restrictive syntax, they can still express problems that are complete for the second level of the polynomial hierarchy.

We also give theoretical evidence that (connection-)guarded programs are promising for solving problems on instances of small treewidth. Indeed, when parameterized by the treewidth of the input, answer set solving is fixed-parameter tractable for fixed guarded programs, and an analogous result holds for connection-guarded programs when the parameter consists of both treewidth and maximum degree. Finally, we showed that the additional dependency on the maximum degree in the case of connection-guarded ASP cannot be dropped (unless $\mathsf{P} = \mathsf{NP}$) as this makes the problem NP-hard.

The results presented in this paper thus shed light on the relationship between treewidth and solving performance and provide us with tools to obtain more efficient ASP encodings. Future work includes the investigation of alternative classes of programs that preserve small treewidth (for instance, taking the concept of tightness (Erdem & Lifschitz, 2003) additionally into account) and extending our research to other width measures from the literature (Eiben, Ganian, & Szeider, 2018; Courcelle, Engelfriet, & Rozenberg, 1991; Gajarský, Lampis, & Ordyniak, 2013).

Acknowledgments

This work is an improved and significantly extended version of the conference publications by Bliem, Moldovan, Morak, and Woltran (2017) and Bliem (2018), and contains additional proofs, constructions, and discussions, based on the PhD thesis of Bliem (2017). It was funded in part by the Austrian Science Fund (FWF) under grant numbers Y698, P30930, and P32830, as well as by the Academy of Finland under grants 276412 and 312662.

References

Alviano, M., Dodaro, C., Leone, N., & Ricca, F. (2015). Advances in WASP. In *Proceedings of LPNMR 2015*, pp. 40–54. Springer.

Amendola, G., Ricca, F., & Truszczynski, M. (2017). Generating hard random boolean formulas and disjunctive logic programs. In Sierra, C., & Bacchus, F. (Eds.), *Proceedings of IJCAI 2017*, pp. 532–538. AAAI Press.

Atserias, A., Fichte, J. K., & Thurley, M. (2011). Clause-learning algorithms with many restarts and bounded-width resolution. *J. Artif. Intell. Res. (JAIR)*, *40*, 353–373.

Bannach, M., & Berndt, S. (2018). Practical access to dynamic programming on tree decompositions. In *Proceedings of ESA 2018*, Vol. 112 of *LIPIcs*, pp. 6:1–6:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Bichler, M., Morak, M., & Woltran, S. (2016). The power of non-ground rules in answer set programming. *Theory and Practice of Logic Programming*, *16*(5-6), 552–569.

Bichler, M., Morak, M., & Woltran, S. (2017). lpopt: A rule optimization tool for answer set programming. In *Revised Selected Papers of LOPSTR 2016*, pp. 114–130. Springer.

Bliem, B. (2017). *Treewidth in Non-Ground Answer Set Solving and Alliance Problems in Graphs*. Ph.D. thesis, Fakultät für Informatik an der Technischen Universität Wien. `http://permalink.obvsg.at/UTW/AC14478683`.

Bliem, B. (2018). ASP programs with groundings of small treewidth. In *Proceedings of FoIKS 2018*, pp. 97–113. Springer.

Bliem, B., Moldovan, M., Morak, M., & Woltran, S. (2017). The impact of treewidth on ASP grounding and solving. In *Proceedings of IJCAI 2017*, pp. 852–858. ijcai.org.

Bodlaender, H. L. (1996). A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing*, *25*(6), 1305–1317.

Bodlaender, H. L., & Koster, A. M. C. A. (2010). Treewidth computations I. upper bounds. *Information and Computation*, *208*(3), 259–275.

Brewka, G., Eiter, T., & Truszczynski, M. (2011). Answer set programming at a glance. *Commun. ACM*, *54*(12), 92–103.

Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Ricca, F., & Schaub, T. (2015). ASP-core-2 input language format. `https://www.mat.unical.it/aspcomp2013/ASPStandardization`.

Călinescu, G., Fernandes, C. G., & Reed, B. A. (2003). Multicuts in unweighted graphs and digraphs with bounded degree and bounded tree-width. *Journal of Algorithms*, *48*(2), 333–359.

Courcelle, B., & Engelfriet, J. (2012). *Graph Structure and Monadic Second-Order Logic – A Language-Theoretic Approach*, Vol. 138 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press.

Courcelle, B., Engelfriet, J., & Rozenberg, G. (1991). Context-free handle-rewriting hypergraph grammars. In *Graph-Grammars and their Application to Computer Science,*

4th International Workshop, Bremen, Germany, March 5–9, 1990, Proceedings, Vol. 532 of *LNCS*, pp. 253–268.

Cygan, M., Fomin, F. V., Kowalik, Ł., Lokshtanov, D., Marx, D., Pilipczuk, M., Pilipczuk, M., & Saurabh, S. (2015). *Parameterized Algorithms*. Springer International Publishing, Cham, Switzerland.

Dantsin, E., Eiter, T., Gottlob, G., & Voronkov, A. (2001). Complexity and expressive power of logic programming. *ACM Comput. Surv.*, *33*(3), 374–425.

Eiben, E., Ganian, R., & Szeider, S. (2018). Solving problems on graphs of high rank-width. *Algorithmica*, *80*(2), 742–771.

Eiter, T., & Gottlob, G. (1995). On the computational cost of disjunctive logic programming: Propositional case. *Ann. Math. Artif. Intell.*, *15*(3-4), 289–323.

Eiter, T., Gottlob, G., & Mannila, H. (1997). Disjunctive datalog. *ACM Trans. Database Syst.*, *22*(3), 364–418.

Elkabani, I., Pontelli, E., & Son, T. C. (2005). Smodels^{a} - A system for computing answer sets of logic programs with aggregates. In *Proceedings of LPNMR 2005*, pp. 427–431. Springer.

Erdem, E., & Lifschitz, V. (2003). Tight logic programs. *Theory and Practice of Logic Programming*, *3*(4-5), 499–518.

Fichte, J. K., Hecher, M., Morak, M., & Woltran, S. (2017). Answer set solving with bounded treewidth revisited. In *Proceedings of LPNMR 2017*, pp. 132–145. Springer.

Fichte, J. K., & Szeider, S. (2015). Backdoors to tractable answer set programming. *Artif. Intell.*, *220*, 64–103.

Fichte, J. K., Kronegger, M., & Woltran, S. (2017). A multiparametric view on answer set programming. In *Proceedings of ASPOCP 2017*. CEUR-WS.org.

Gajarský, J., Lampis, M., & Ordyniak, S. (2013). Parameterized algorithms for modular-width. In *Proceedings of IPEC*, Vol. 8246 of *LNCS*, pp. 163–176. Springer.

Gebser, M., Kaminski, R., Kaufmann, B., Lindauer, M., Ostrowski, M., Romero, J., Schaub, T., & Thiele, S. (2015). Potassco user guide. `https://sourceforge.net/projects/potassco/files/guide/`.

Gebser, M., Kaminski, R., Kaufmann, B., & Schaub, T. (2012). *Answer Set Solving in Practice*. Morgan & Claypool.

Gebser, M., Kaufmann, B., Kaminski, R., Ostrowski, M., Schaub, T., & Schneider, M. T. (2011). Potassco: The Potsdam answer set solving collection. *AI Commun.*, *24*(2), 107–124.

Gebser, M., Kaufmann, B., & Schaub, T. (2012). Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, *187*, 52–89.

Gelfond, M., & Lifschitz, V. (1988). The stable model semantics for logic programming. In *Proc. ICLP*, pp. 1070–1080.

Gelfond, M., & Lifschitz, V. (1991). Classical negation in logic programs and disjunctive databases. *New Generation Computing*, *9*(3/4), 365–386.

Gottlob, G., Grädel, E., & Veith, H. (2002). Datalog LITE: a deductive query language with linear time model checking. *ACM Trans. Comput. Log.*, *3*(1), 42–79.

Gottlob, G., Pichler, R., & Wei, F. (2010). Bounded treewidth as a key to tractability of knowledge representation and reasoning. *Artif. Intell.*, *174*(1), 105–132.

Jakl, M., Pichler, R., & Woltran, S. (2009). Answer-set programming with bounded treewidth. In *Proceedings of IJCAI 2009*, pp. 816–822.

Kloks, T. (1994). *Treewidth, Computations and Approximations*, Vol. 842 of *Lecture Notes in Computer Science*. Springer.

Knop, D., Koutecký, M., Masarík, T., & Toufar, T. (2017). Simplified algorithmic metatheorems beyond MSO: treewidth and neighborhood diversity. In *Graph-Theoretic Concepts in Computer Science - 43rd International Workshop, WG. Revised Selected Papers*, pp. 344–357.

Kreutzer, S. (2012). On the parameterized intractability of monadic second-order logic. *Logical Methods in Computer Science*, *8*(1).

Langer, A., Reidl, F., Rossmanith, P., & Sikdar, S. (2012). Evaluation of an mso-solver. In Bader, D. A., & Mutzel, P. (Eds.), *Proceedings of ALENEX 2012*, pp. 55–63. SIAM / Omnipress.

Langer, A., Reidl, F., Rossmanith, P., & Sikdar, S. (2014). Practical algorithms for MSO model-checking on tree-decomposable graphs. *Computer Science Review*, *13-14*, 39–74.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., & Scarcello, F. (2006). The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, *7*(3), 499–562.

Marek, V. W., & Truszczyński, M. (1999). Stable models – an alternative logic programming paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*, pp. 375–398. Springer.

Matoušek, J., & Thomas, R. (1992). On the complexity of finding iso- and other morphisms for partial k-trees. *Discrete Mathematics*, *108*(1-3), 343–364.

Morak, M., Pichler, R., Rümmele, S., & Woltran, S. (2010). A dynamic-programming based ASP-solver. In *Proceedings of JELIA 2010*, pp. 369–372. Springer.

Niedermeier, R. (2006). *Invitation to Fixed-Parameter Algorithms*, Vol. 31 of *Oxford Lecture Series in Mathematics and its Applications*. Oxford University Press.

Peters, D. (2017). Precise complexity of the core in dichotomous and additive hedonic games. In *Proceedings of ADT 2017*, pp. 214–227. Springer.

Pichler, R., Rümmele, S., Szeider, S., & Woltran, S. (2014). Tractable answer-set programming with weight constraints: bounded treewidth is not enough. *Theory and Practice of Logic Programming*, *14*(2), 141–164.

Robertson, N., & Seymour, P. D. (1986). Graph minors. II. algorithmic aspects of treewidth. *Journal of Algorithms*, *7*(3), 309–322.

Selman, B., Mitchell, D. G., & Levesque, H. J. (1996). Generating hard satisfiability problems. *Artif. Intell.*, *81*(1-2), 17–29.

Stockmeyer, L. J. (1974). *The Complexity of Decision Problems in Automata Theory*. Ph.D. thesis, Department of Electrical Engineering, MIT.

Szeider, S. (2003). On fixed-parameter tractable parameterizations of SAT. In *Proceedings of SAT 2003*, pp. 188–202. Springer.

Truszczyński, M. (2011). Trichotomy and dichotomy results on the complexity of reasoning with disjunctive logic programs. *Theory and Practice of Logic Programming*, *11*(6), 881–904.

Tseitin, G. S. (1983). On the complexity of derivation in propositional calculus. In *Automation of Reasoning*, pp. 466–483. Springer.

Urquhart, A. (1987). Hard examples for resolution. *JACM*, *34*(1), 209–219.

Vardi, M. Y. (1982). The complexity of relational query languages (extended abstract). In Lewis, H. R., Simons, B. B., Burkhard, W. A., & Landweber, L. H. (Eds.), *Proceedings of STOC 1982*, pp. 137–146. ACM.

Wen, L., Wang, K., Shen, Y., & Lin, F. (2016). A model for phase transition of random answer-set programs. *ACM Trans. Comput. Log.*, *17*(3), 22:1–22:34.

Zhao, Y., & Lin, F. (2003). Answer set programming phase transition: A study on randomly generated programs. In *Proc. ICLP*, pp. 239–253. Springer.